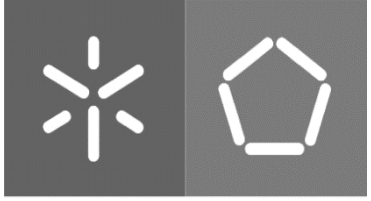
Universidade do Minho
Escola de Engenharia

Nuno André Costa da Silva

Hardware IPC for a TrustZone-assisted Hypervisor

Outubro de 2018



Universidade do Minho
Escola de Engenharia

Nuno André Costa da Silva

Hardware IPC for a TrustZone-assisted Hypervisor

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do

Doutor Sandro Pinto
Doutor Adriano Tavares

Outubro de 2018

Acknowledgements

First, I would like to thank my advisor Dr. Sandro Pinto for not only providing me with this thesis' innovative concept and helping me extensively throughout its development, but for also allowing me to make my first scientific contribution and subsequent participation in my first conference, as well as establishing connections that provided me with my first engineering job.

I would also like to thank Dr Adriano Tavares, my co-advisor and Embedded Systems master's professor, for supplying me with an endless amount of knowledge, as well as pushing me to thrive and be confident with my knowledge and skills.

Also, thanks to my colleagues who worked sleepless hours alongside me in the lab that, for a few months, became more of a "home" than my house. Not only that, but all of them, Ailton Lopes, Ângelo Ribeiro, Franciso Petrucci, Hugo Araújo, José Ribeiro, José Silva, Pedro Machado, Ricardo Roriz, and Sérgio Pereira, helped me solve the most tiring of problems while keeping me "sane", and for that, I thank you. Thanks to João Alves and André Oliviera for being my IPC mentors, José Martins for helping me with the more difficult problems that my lab colleagues couldn't help me solve, and Miguel Silva for keeping my (mild) OCD in check.

Finally, I would like to thank my Mom and Dad, Gabriela and Filipe, you both know what you mean to me and I hope to always make you proud; my brother Jorge and cousins Ricardo and Carolina, for helping me proof-read this document; my grandparents and the rest of my family, for helping and believing in me. All of you were greatly encouraging and crucial during the journey that was this engineering course, and for that, I cannot thank you enough.

Abstract

In this modern era ruled by technology and the IoT (Internet of Things), embedded systems have an ubiquitous presence in our daily lives. Although they do differ from each other in their functionalities and end-purpose, they all share the same basic requirements: safety and security. Whether in a non-critical system such as a smartphone, or a critical one, like an electronic control unit of any modern vehicle, these requirements must always be fulfilled in order to accomplish a reliable and trust-worthy system.

One well-established technology to address this problem is virtualization. It provides isolation by encapsulating each subsystem in separate Virtual-Machines (VMs), while also enabling the sharing of hardware resources. However, these isolated subsystems may still need to communicate with each other. Inter-Process Communication is present in most OSes' stacks, representing a crucial part of it, which allows, through a myriad of different mechanisms, communication between tasks. In a virtualized system, Inter-Partition Communication mechanisms implement the communication between the different subsystems referenced above.

TrustZone technology has been in the forefront of hardware-assisted security and it has been explored for virtualization purposes, since natively it provides separation between two execution worlds while enforcing, by design, different privilege to these execution worlds. LTZVisor, an open-source lightweight TrustZone-assisted hypervisor, emerged as a way of providing a platform for exploring how TrustZone can be exploited to assist virtualization. Its IPC mechanism, TZ-VirtIO, constitutes a standard virtual I/O approach for achieving communication between the OSes, but some overhead is caused by the introduction of the mechanism. Hardware-based solutions are yet to be explored with this solution, which could bring performance and security benefits while diminishing overhead.

Attending the reasons mentioned above, hTZ-VirtIO was developed as a way to explore the offloading of the software-based communication mechanism of the LTZVisor to hardware-based mechanisms.

Resumo

Atualmente, onde a tecnologia e a Internet das Coisas (IoT) dominam a sociedade, os sistemas embebidos são omnipresentes no nosso dia-a-dia, e embora possam diferir entre as funcionalidades e objetivos finais, todos partilham os mesmos requisitos básicos. Seja um sistema não crítico, como um smartphone, ou um sistema crítico, como uma unidade de controlo de um veículo moderno, estes requisitos devem ser cumpridos de maneira a se obter um sistema confiável.

Uma tecnologia bem estabelecida para resolver este problema é a virtualização. Esta abordagem providencia isolamento através do encapsulamento de subsistemas em máquinas virtuais separadas, além de permitir a partilha de recursos de hardware. No entanto, estes subsistemas isolados podem ter a necessidade de comunicar entre si. Comunicação entre tarefas está presente na maioria das pilhas de software de qualquer sistema e representa uma parte crucial dos mesmos. Num sistema virtualizado, os mecanismos de comunicação entre-partições implementam a comunicação entre os diferentes subsistemas mencionados acima.

A tecnologia TrustZone tem estado na vanguarda da segurança assistida por hardware, e tem sido explorada na implementação de sistemas virtualizados, visto que permite nativamente a separação entre dois mundos de execução, e impondo ao mesmo tempo, por *design*, privilégios diferentes a esses mundos de execução. O LTZVisor, um hypervisor em código-aberto de baixo *overhead* assistido por TrustZone, surgiu como uma forma de fornecer uma plataforma que permite a exploração da TrustZone como tecnologia de assistência a virtualização. O TZ-VirtIO, mecanismo de comunicação do LTZVisor, constitui uma abordagem padrão de E/S virtuais, para permitir comunicação entre os sistemas operativos. No entanto, a introdução deste mecanismo provoca sobrecarga sobre o hypervisor. Soluções baseadas em hardware para o TZ-VirtIO ainda não foram exploradas, e podem trazer benefícios de desempenho e segurança, e diminuir a sobrecarga.

Atendendo às razões mencionadas acima, o hTZ-VirtIO foi desenvolvido como uma maneira de explorar a migração do mecanismo de comunicação baseado em software do LTZVisor para mecanismos baseados em hardware.

Contents

List of Figures

List of Listings

Glossary

ABI	Application Binary Interface
ACP	Accelerator Coherency Port
ACTLR	Auxiliary Control Register
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
APU	Application Processor Unit
ASIC	Aplication Specific Integratedd Circuit
AXI	Advanced Extensible Interface
CE	Consumer Electronics
CPU	Central Processing Unit
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
DoS	Denial-of-Service
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processing
FIQ	Fast Interrupt Request
FPGA	Field Programmable Gate Array
GIC	Generic Interrupt Controller
GPIO	General-Purpose Input/Output
GPOS	General-Purpose Operating System
I/O	Input/Output
IDE	Integrated Development Environment
IOMMU	Input/Output Memory Management Unit
IoT	Internet of Things
IPC	Inter-Partition Communication
IPI	Inter-Processor Interrupt
IRQ	Interrupt Request
LoC	Lines-of-Code

LR	Linker Register
LTZVisor	Lightweight TrustZone-assisted Hypervisor
MMU	Memory Management Unit
MPMU	Message Passing Management Unit
OCM	On-Chip Memory
OS	Operating System
PL	Programmable Logic
PS	Processing System
RAM	Random-Access Memory
ROM	Read-Only Memory
RTOS	Real-Time Operating System
SCTLR	System Control Register
SCU	Snoop Control Unit
SLCR	System-Level Control registers
SMC	Secure Monitor Call
SoC	System on-Chip
SP	Stack Pointer
SPSR	Saved Program Status Register
SRAM	Static Random-Access Memory
TCB	Trusted Computing Base
TZ	TrustZone
TZASC	TrustZone Address Space Controller
TZMA	TrustZone Memory Adapter
TZPC	TrustZone Protection Controller
VM	Virtual Machine
VMCB	Virtual Machine Control Block
VMM	Virtual Machine Monitor
XSDK	Xilinx Software Development Kit

1. Introduction

More and more, society is becoming more technologically advanced, with computers being part of almost all common devices and services. Although people are aware of them and interact directly with some (personal computers, smartphones, automotive infotainment systems, etc,), most computers are small and unnoticeable (smart home appliances, barcode scanners, automotive electronic control units, etc.), and are becoming ubiquitous in our daily lives. These are usually referred to as embedded systems. Furthermore, every category of smart devices need, increasingly, to be connected with each other, either with other surrounding systems, or with devices all around the globe. This has led to the rise of the now infamous concept of IoT (Internet of Things).

Far from just a theoretical concept however, IoT devices have been increasing in both quantity and complexity while following trends towards small-form factor (i.e., reduced in size). In fact, system engineers aim to follow an assortment of standardized metrics, the SWaP-C (Size, Weight, Power and Cost) metrics, when designing embedded systems. The goal should be to keep these metrics as low as possible without compromising performance and the effectiveness of the system. Accomplishing all of these requirements might be a difficult task, but its intricacies still quiver when faced with the added complication of maintaining device integrity, especially in critical systems, where failure is not an option. Basically, divided into safety and security concerns, these critical embedded systems must preserve their integrity throughout execution, and this means that these devices must be safe and secure. This is especially relevant when considering that IoT end-devices are burdened with the task of exchanging great quantities of security-critical and privacy-sensitive data all over the globe, and are also much more prone to cyber-attacks [PGP+17].

So, because the demand for smaller, more complex and more powerful system was ever-growing, coupled with the need for security in these same systems, there was also an obvious need for technological evolution to support such trend. To

achieve embedded systems with such characteristics, researchers turned to virtualization [Hei08, Kai09]. Because virtualization enables concurrent execution of multiple virtual machines on the same hardware platform while also enabling sharing of hardware resources, not only does it prove to be a great candidate for the reduction of the SWaP-C metrics, but it also tackles problems associated with security threats, providing security by isolation.

First off, the SWaP-C benefits of consolidating multiple systems in a single platform is evident, as eliminating excess of federated hardware resources will greatly reduce all of these metrics. Also, in a virtualization environment, although the various partitions share the same hardware resources, they are segregated from each other, which helps to settle some security concerns. It is not easy however, to achieve an effective system that consolidates what were previously multiple systems in one hardware platform and still keep the same performance and real-time capabilities without compromising security. This kept researchers trying to find new strategies that would be able to achieve the desired metrics while keeping up with demand. Naturally, solutions evolved to make use of the underlying hardware platform to help the software layer, both in performance and security departments.

The first adopted strategy was hardware-assisted virtualization, which quickly became the norm in virtualized embedded systems solutions. Leading SoC manufacturers, mainly ARM and Intel, started incorporating hardware virtualization technology (ARM's Virtualization Extensions and Intel's Virtualization Technology) in their SoCs. This enabled system designers to make use of the underlying hardware platform to greatly reduce software virtualization overhead. Furthermore, ARM's TrustZone technology, which in fact is not a virtualization extension per se, but rather a security extension technology, quickly became the preferred solution amongst researchers in the embedded systems domain. Its, by design, division into secure and non-secure worlds led to the development of a myriad of dual-guest hypervisors which made use of TrustZone's hardware separation to completely isolate the trusted from non-trusted components, forbidding the non-trusted, less privileged components from affecting the most-critical trusted partitions of the system [SHT10, KLJ+13, LCP+17, POP+17]. Furthermore, other solution that was followed to achieve performance improvements in these embedded IoT devices, was hardware offloading [GPG+15]. By offloading some of the more overhead-heavy software components to an FPGA SoC, designers were able to get more efficient systems with the small disadvantage of additional hardware costs. Either by being used as a standalone hardware offloading technique [GSP+16], or

as an addition to an already hardware-assisted virtualization system [XPN15a], hardware offloading has been gaining traction as one of the most used strategies for embedded systems solutions.

One of these hardware-assisted implementations is the in-house LTZVisor, an open-source, Lightweight, TrustZone-assisted hypervisor capable of running in a dual-OS configuration, usually an OS with real-time capabilities in the secure world for the time-critical tasks, and a General-Purpose OS for user interaction. Although these partitions need to be completely isolated, they sometimes need to transfer information between them. As such, LTZVisor implements a secure, shared-memory communication mechanism, TZ-VirtIO [OMC+18], thus providing a way for OSes to share data between them.

To demonstrate how a communication channel might be useful in these kinds of systems, we could make an use-case out of a virtualized embedded system for a system comprised of an automotive electronic control unit and an infotainment dashboard. Figure 1.1 represents, at a very basic level, how LTZVisor and TZ-VirtIO handle communication between two OSes in the same hardware platform. In this situation, the RTOS will handle the critical portion of the system, collecting data from the electronic control unit while managing some important hardware mechanisms. Meanwhile, the GPOS will handle the infotainment system, which includes displaying some important information collected by the electronic control unit. So, because these systems are virtualized and not aware of each other, there is the need for a communication strategy between the two, and the regular networking over-the-air connection is not an option for a system with this level of criticality. This is where TZ-VirtIO comes in as a secure communication channel between the two, allowing for systems to share data through the hardware platform while preventing the untrusted partition from affecting the trusted, secure one.

This strategy however, does bring a lot of software overhead to the system, overhead which might be fatal for the real-time requirements of the system. For instance, assuming a situation where the RTOS partition of the system is both handling the breaking system while collecting and sending data to the GPOS side, the CPU should not be stalled while transferring data, as it will keep the real-time partition from acting accordingly when the driver needs to act on the brakes, as it could obviously jeopardise the safety of the vehicle's passengers. One solution for this problem, could be the aforementioned hardware offloading of the transferring mechanism. If the data transfer is executed in hardware, the CPU would be freed for more critical tasks, and thus the safety of the system would not be compromised. Furthermore, if migrated to hardware, the data throughput

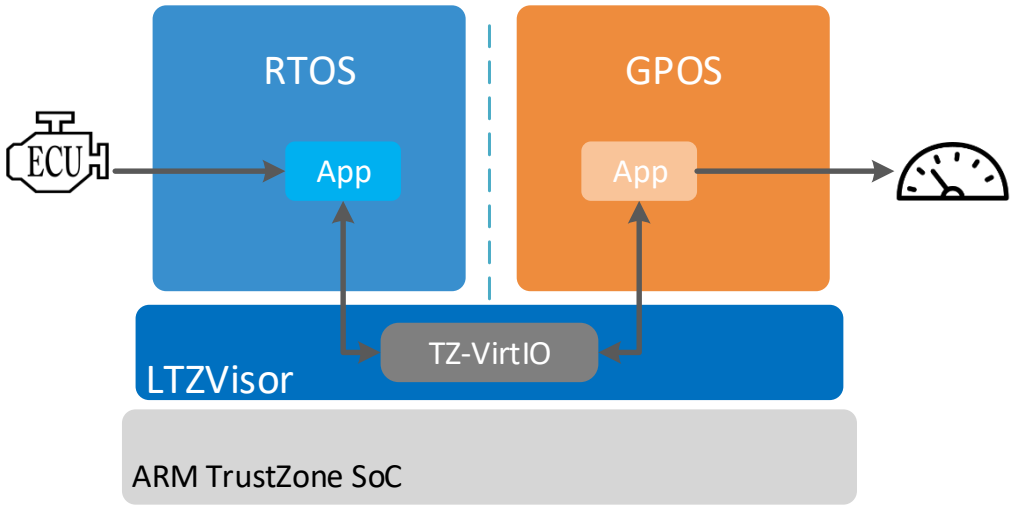


Figure 1.1: General Automotive Communication Use-Case

could be much greater than that of a software-based solution and thus it also improves overall system performance.

1.1 Goals

To better assert if the aforementioned TZ-VirtIO hardware offloading would be an effective alternative, this thesis proposes the evaluation of the advantages and disadvantages of modifying the current mechanism to include an hardware-based approach. As such the main goal of this thesis should be the implementation of dedicated hardware mechanisms capable of replacing the TZ-VirtIO's data channel, and the subsequent comparison between the implemented strategies.

Firstly, an extensive study on the current State of the Art must be carried out, alongside an in-depth familiarization with the used platform and tools necessary for the implementation of this project. Thus, the proposed methodology for this project will be as follows:

1. In-depth State of the Art review, analysing the many previously used strategies and implementations, as well as the theoretical fundamentals for this thesis. This research should be carried on throughout the implementation phase, as the State of the Art could be continuously updated.

2. Extensive exploration of the used platform and tools, including the Zynq-7000 SoC, ARM's TrustZone technology, LTZVisor and TZ-VirtIO, as well as a painstaking understanding of the used hardware mechanisms.

3. Implementation of the different proposed hardware solutions, including its integration in the complete system.

4. Extensive evaluation and comparison of the achieved solutions.

5. Detailed writing of this document, including the whole of the aforementioned efforts, as well as the thesis' findings.

Further chapters will delve deeper into each phase of the project.

1.2 Document Structure

In this section will be presented how this document's chapters are structured, as well as their contents and purpose. The structure is as follows:

1. **Introduction** (current chapter): explains the motivation around this thesis, as well as its main goals.

2. **Background, Context, and State of the Art**: covers the theoretical fundamentals and other works related to this thesis. It covers TrustZone technology, virtualization and hardware-assisted virtualization, and explains Inter-Partition Communication mechanisms, followed by relevant implementations that make use of these fundamentals.

3. **Platform and Tools**: goes over the platform and tools used in this project, including both hardware and software platforms, and diving deeper into the more relevant topics.

4. **hTZ-VirtIO: Hardware IPC Mechanism for LTZVisor**: demonstrates the implementation of the various components, including the hardware modules and their integration in the software platform.

5. **Evaluation**: presents the ultimate findings and comparisons, as well as a discussion on them.

6. **Conclusion**: final conclusion of the document, discussing its relevancy to the community after the complete research and implementation is terminated. Also presents some future work that might be relevant for further research.

2. Background, Context, and State of the Art

This chapter will present the fundamental concepts needed for the development of this project, as well as some existing related work relevant to the thesis' theme. It starts by explaining and contextualising virtualization, with focus on the embedded computing domain, followed by a breakdown on how Inter-Partition Communication is relevant to any embedded system, virtualized or not. The "Related Work" section presents several IPC mechanisms that are somehow related to the developed communication mechanism.

2.1 Background

This section will explore the background concepts and theoretical fundamentals that are relevant to this thesis, starting by explaining classical virtualization techniques, followed by Hardware-assisted, and more specifically TrustZone-assisted, virtualization. Secondly, IPC is introduced, focusing on its importance to embedded systems design alongside various mechanisms, including some hardware-based ones. Lastly, the VirtIO system is briefly explained.

2.1.1 Virtualization

For the everyday computer user, the word "virtualization" is, almost always, attached solely to the notion of a computer program atop a General-Purpose Operating System (GPOS), that is, capable of emulating another GPOS. Unbeknownst to the common user, virtualization technology is being used in a myriad of electronic devices from their quotidian, although at a much deeper abstraction level from that of a normal user-level virtualization software. Rather, the original and more common use of virtualization consists on an abstraction layer directly above the hardware platform (i.e. system-level virtualization).

For either application, this abstraction layer, commonly referred to as Virtual Machine Monitor (VMM) or hypervisor, is responsible for multiplexing the resources available to one or more Virtual-Machines (VM). This also allows the VMs, acting as Guests, to run in an isolated environment, while the hypervisor, acting as the Host, arbitrates the VMs' scheduling policy and mediates hardware access, as it runs in an higher privilege level than the Virtual Machines running on top of it. These Guests can be either full fledged OSes or bare-metal applications and in both cases, they may or may not have to be ported to run successfully. This gives us the distinction between full virtualization, where the Guests' code needs no alterations or additions and is unaware of running atop an hypervisor, and paravirtualization, where a Guest is ported to run in top of a specific hypervisor while being aware that is doing so. Both techniques have their advantages, being the most obvious, the easiness of incorporation of Guests on the full virtualization strategy, and the performance boost from having hypervisor compliant APIs on paravirtualization's side.

Depending on the position of the virtualization layer in the software stack, it is possible to distinguish two types of hypervisor, represented in Figure 2.1:

- **Type-1**, or bare-metal hypervisors, have direct access to the hardware layer and manage the execution permissions of every system component, meaning all hardware accesses are to be mediated and controlled by the VMM. As a consequence, the performance degradation of guest OSes will only be influenced by the performance of the hypervisor itself, making this type of hypervisor more suited to systems that must meet time constraints and also, the usual choice for an embedded system application.

- **Type-2**, or hosted hypervisors, do not run directly above the hardware layer. Instead, they run over a OS that is already executing. This type of VMM usually does not have permissions to access and perform any operation on the hardware directly, since those responsibilities usually rest in the system software that runs below the VMM, usually resulting in lower performance ratings compared to type-1 hypervisors.

From this point forward throughout this document, we refer to an "Hypervisor" to convey a type-1 hypervisor, as this is the type of virtualization layer used in this thesis' work.

To properly partition a system to support concurrent execution of multiple VMs, a basic set of efforts must be achieved [BDF+03]. Firstly, the virtual machines must be isolated from each other, as they should not affect or be affected

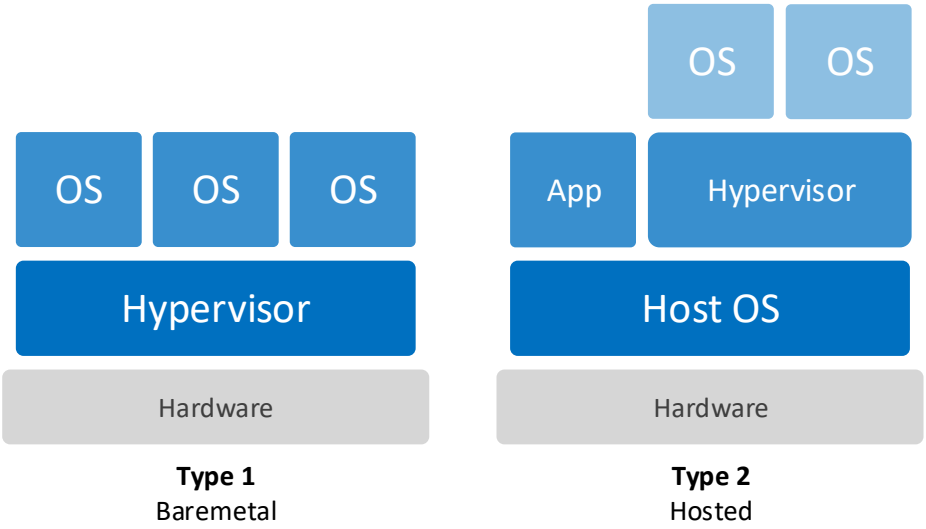


Figure 2.1: Hypervisor Types

by one another. This rule should be more relevant in the case of a critical system, where for instance, one of the VMs is a Real-Time Operating System (RTOS) which must meet real-time constraints. More on this subject will be explored ahead. The hypervisor should also be able to run a variety of Operating Systems, depending on the heterogeneity of the desired system, although this may not be important in most low-end embedded system applications. Lastly, there should be a low performance overhead induced by the introduction of the virtualization layer.

This offers great advantages for modern embedded systems, as the smaller form-factor, achieved by the consolidation of multiple systems in a single hardware platform, is a crucial part of any of these systems [Hei11, GGS+14]. Furthermore, and as mentioned before, the ability to conjugate a general-purpose operating system for user-friendly applications, with a real-time operating system for carrying out time-critical tasks (e.g. infotainment/automotive control systems, smartphones), is greatly assisted by virtualization technology [Hei08, ISM09]. For critical embedded systems though, there is another major advantage brought on by virtualization, the safety and security achieved by its spatial and temporal isolation. Isolation achieved by a virtualization solution is more greatly imposed, compared to any typical OS, as it is accomplished by design [Hei08, Kai08].

Considering the benefits of virtualization, it is no surprise that, along with the growth in functionality in consumer electronics (CE) [Hei09], or the growth in complexity, criticality and quantity of spatial/aeronautical systems [PPG+17],

or the rise of embedded automotive solutions merging infotainment and critical systems [RM14], there has also been a rise in embedded solutions making use of virtualization technology for their system, as they all share the same interest: consolidating their systems while maintaining isolation.

Virtualization however, comes with its own drawbacks, being the unavoidable performance overhead the biggest one amongst problems of embedded systems' virtualization. Another hindrance of classical virtualization used in critical embedded systems is the creation of a single point of failure, the VMM, which alongside its usual complexity and largeness of its trusted computing based (TCB), makes it both a easy target for hackers while also extremely bug prominent [GGS+14].

To solve these problems, a myriad of different solutions have been proposed: the first option adopted was to reduce the TCB size, by simply implementing either lightweight or microkernel versions of hypervisors, as seen in [HL10, ISM09]; another popular option was to make use of virtualization hardware extensions like Intel's Virtualization Technology [UNR+05, NSL+06], ARM's Virtualization Extensions [MN11] and TrustZone (TZ) [ARM09] technology, as seen in [DN14, YSA+06, PTM16]; many solutions also applied a combination of both hardware extensions and TCB reduction, as seen in [PPG+17, MAC+17]; some solutions implemented hardware acceleration [XPN15b] and even going as far as experimenting with dynamic partial reconfiguration-assisted hypervisors [XPN15a].

2.1.2 ARM TrustZone

As this thesis project uses an hypervisor with TrustZone-assisted virtualization as its core, we should delve deeper in the TrustZone architecture and how it works.

ARM TrustZone [ARM09] technology refers to the security extensions introduced by ARM with its ARMv6 architecture in their Cortex-A processors [PS], but which recently made way to the Cortex-M line of processors, beginning with the ARMv8-M architecture. TrustZone technology for both lines of processors share the same basic design features, the only difference being the optimization for microcontrolers and low-power applications for the M family of processors [MAC+17]. From this point on, this thesis will focus on TrustZone for the Cortex-A family of processors.

The simplest way to view TrustZone technology is as a dual-virtual system, where all its physical resources are partitioned into two isolated virtualized execution environments (Figure 2.2). This separation into secure and non-secure world happens on almost every aspect of the TrustZone-enabled System-on-Chip's architecture, with greater relevance at the processor architecture level. The secure and

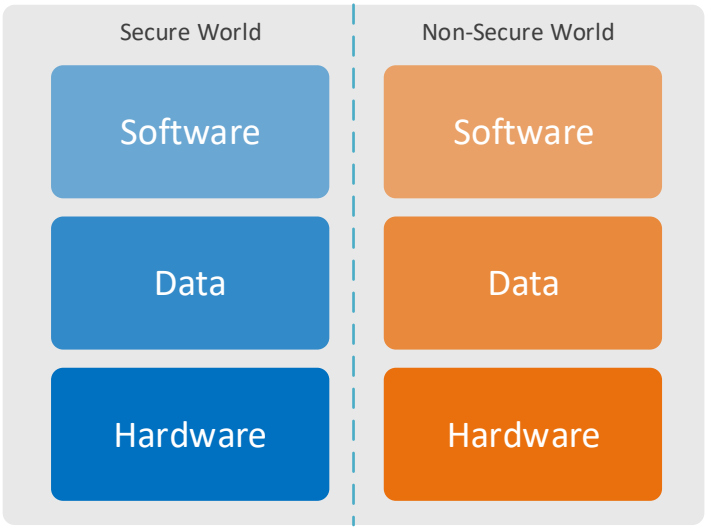


Figure 2.2: ARM TrustZone Technology

non-secure world separation is indicated by a TrustZone exclusive 33rd processor bit, the NS (Non-Secure) bit, which also extends to the memory and remaining peripherals buses. There is also an extra processor mode, monitor mode, added by TrustZone, as a way to ensure preservation of the processor state during the world switch. This mode does not depend on the state of the NS processor bit, as it runs with an higher privilege level than that of the other processor modes, and thus always considered secure [MAC+17].

To bridge the software stacks of both worlds, as the processor can only run in one world at a time, a new privileged instruction was added, the SMC (Secure Monitor Call) instruction. There is also the possibility of entering monitor mode by configuring it to handle interruptions and abort exceptions in the secure world. To further ensure isolation, TrustZone specifies one set of exception vector table for the normal world, one for the secure world, and another for the monitor mode. To handle both secure and non-secure interrupts, the Generic Interrupt Controller (GIC) provides both secure and non-secure prioritized interrupt sources, while also having the ability to prioritize secure interrupts over non-secure interrupts. Furthermore, TrustZone banks a set of special registers to enforce secure and non-secure world isolation, such as a number of System Control Coprocessor (CP15) registers, forbidding the non-secure world of accessing these registers, or at least only being able to access them under the supervision of the secure world.

As mentioned before, the NS bit extends to memory and devices as to maintain world separation on access to these components. This however, is not enough to ensure strong isolation between the worlds and as such, a number of hardware peripherals are added. These include the TrustZone Address Space Controller

(TZASC) and the TrustZone Memory Adapter (TZMA) to extend TrustZone security to the memory, and the TrustZone Protection Controller (TZPC) to extend TrustZone functionalities to other system devices.

The TZASC, only programmable from the secure world, is able to partition the DRAM into different, secure or non-secure, memory regions. TrustZone allows, by design, the secure side to access the non-secure memory regions while forbidding the opposite operation. TZMA also provides the same memory separation features but for external ROM and SRAM. The Memory Management Unit (MMU) is also TrustZone compliant, providing two distinct MMU interfaces and thus enabling each world to have a local set of virtual-to-physical memory address translation tables. TrustZone security also extends to cache-level, since processor's caches include an additional security tag to signal in which state the processor accesses the memory. Furthermore, device security status can be dynamically changed during runtime via the TZPC, which supplies control signals to the various devices. All three of these TrustZone hardware controllers are of optional use and implementation-specific.

2.1.3 TrustZone-assisted Virtualization

Hardware-assisted virtualization is no news to the embedded system world, as hardware support for virtualization is one of the more common solutions for embedded, virtualization applications. In previous sections, we mentioned hardware extensions from Intel and ARM, both of which have been researched extensively and, although they all have their benefits, ARM's TrustZone technology has been the main focus among researchers, despite the fact of TrustZone extensions not being virtualization-oriented, but mainly security-oriented (Figure 2.3).

Some of TrustZone's features however, are similar to other hardware-assisted virtualization techniques. In fact, TrustZone's higher privilege mode, unprivileged modes for guest OSes, and the ability to have full control over the exception system, all mean that TrustZone is a suitable candidate for an hardware-assisted virtualization strategy. TrustZone however, does not permit a classical full-virtualization approach. By only featuring memory support through TrustZone peripherals, such as the TZASC and the TZMA, and not providing two-level address translation, it demands guests to be specifically compiled to run in their designed memory segments. Their code however, does not need to be modified, and thus, still leans more to a full-virtualization strategy, than to a paravirtualization one. This and other smaller disadvantages, do not alienate TrustZone from

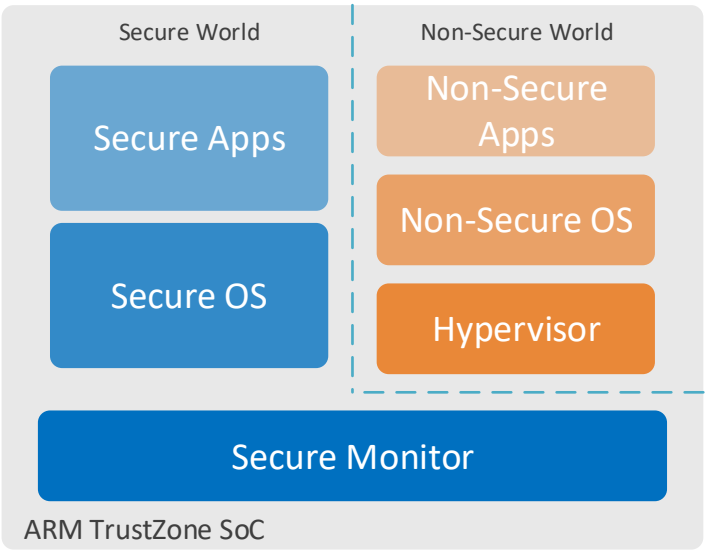


Figure 2.3: ARM TrustZone Technology

being a viable solution for embedded use-cases, especially considering the ubiquity of ARM platforms in everyday embedded devices.

Some of the most famous dual-guest TrustZone-based hypervisors include:

- **SafeG** makes use of TrustZone to implement a dual-OS configuration. It consists of a RTOS in the secure world for time-critical tasks and a GPOS in the non-secure world. SafeG also includes a health monitoring mechanism, a secure device sharing mechanism, and a cyclic and priority-based scheduling approach [SHT10].
- **SASP** or Secure Automotive Software Platform, also implements a dual-OS, RTOS and GPOS, configuration with focus on secure device access [KLJ+13].
- **VOSYSmonitor** like the others implements dual-OS virtualization but, unlike the other, over the newer ARMv8-A architecture. [LCP+17].
- Finally, **LTZVisor** also features a dual-OS configuration but, unlike the others, is an open-source hypervisor. More will be explained about this hypervisor as this will be the platform used in this thesis' work [PPG+17].

Some solutions go further and utilize TrustZone in a multi-guest (more than two) hypervisor application. hypervisors like the RTZVisor and its successor, the μRTZVisor, run multiple guest OSes once at a time on the Non-Secure side, relying on the MMU's support for 2-level address translation, to map guest-virtual to guest-physical addresses and then guest-physical to host-physical addresses, which is crucial to maintain isolation between partitions [PTM16]. The latter

derives from the first, making use of a microkernel approach and capability-based IPC mechanism as key differences [MAC+17].

Furthermore, others make use of a FPGA SoC's hardware extensibility along with TrustZone hardware extensions to maximize virtualization performance while reducing software overhead. The greatest example is Xia's Mini-NOVA [XPN15a], which, like the μRTZVisor, makes use of a microkernel approach while adding dynamic partial reconfiguration (DPR) to implement dynamically reconfigurable hardware tasks.

2.1.4 IPC

Traditionally, IPC refers to the inter-process communication mechanism in any modern OS. It constitutes a crucial part of any OS as it allows processes to share data between each other. Although ubiquitous and sharing, more or less, the same final purpose, IPC mechanisms differ greatly in many core characteristics, depending on the use-case application, hardware and software resources available, and many other system attributes. In fact, in a mircokernel-oriented OS, the IPC mechanism is of even greater importance, since the out-of-kernel components rely on IPC to share information between each other [HEK+07, Lie93b, SFS96].

On a virtualization environment, there may also be the need for a communication mechanism. Although this mechanism is usually still referred to as IPC, it actually stands for inter-partition communication. Other denominations include inter-VM communication and inter-domain communication. Usually, communication between virtual-machines is done through a TCP/IP network stack. This method however, does not represent a practical solution for an embedded system hypervisor, where most communication is critical and with the need of being executed in the shortest amount of time possible. This makes TCP/IP communication unpratical for this kind of scenario, as it induces too much of a performance overhead on the system, having the possibility of being as high as the communication cost of transferring data between VMs located on separate physical machines [RLZ+16]. Instead, the best strategy should be to make use of the underlying hypervisor layer to manage the communication between partitions, while still maintaining communication abstraction for the VMs (Figure 2.4).

To maintain separation between VMs, and because the basic rules of virtualization tell us that there must be isolation between partitions, the IPC mechanism must be responsible for guaranteeing that Guests are isolated from each other while sharing data between them. This is of greater relevance in security critical scenarios, as an attack on one of the VMs should not affect the other partitions,

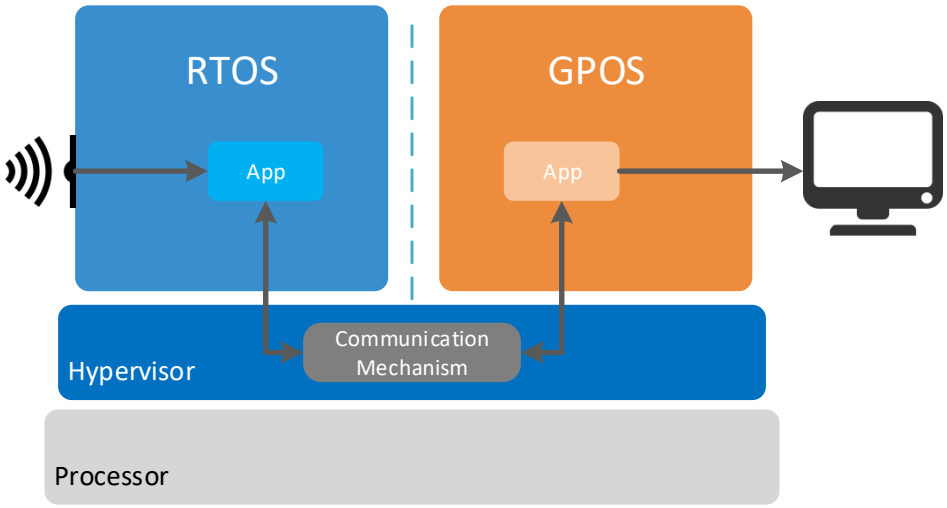


Figure 2.4: General Dual-OS Communication Use-Case

or the system itself. [TGB+08] presents us with a basic set of rules to follow when designing a trustworthy, dependable and secure IPC mechanism, including: reliable message delivery and overall integrity, independent IPC calls, impossible to snoop message traffic, etc.

2.1.4.1 IPC Mechanisms

IPC mechanisms used in virtualization solutions are, with some minor differences, equivalent to those present in most OSes. They can be classified in relation to their synchronism, privilege level, and overall message transfer mechanism. Naturally, these differences translate into performance, security and memory footprint metrics fluctuations, making them ideal for different situations. In this section we will explore this mechanisms and their advantages and disadvantages for the different scenarios.

Regarding synchronism, communication can be divided into:

- **Synchronous communication** or blocking communication: Each partition must wait for the other partition's response, so that they meet at a given moment to proceed with communication. An example use-case could be: a partition sending a message is in a sending state, specifying the message to be transferred, while the receiver partition is prepared for the acceptance of a new message, specifying the incoming messages buffer. If sender and receiver must be synchronized to complete the message transfer, one of the partitions must be blocked while waiting for the interdependent operation.

This method usually entails performance and resource management benefits, since the message transfer is made with the least amount of copies possible, reducing message latency and kernel buffering [EH13]. Regarding security, a synchronous IPC design should take in consideration some extra factors, as these type of mechanisms are prone to Denial-of-Service (more commonly known as DoS) attacks, or even unintended deadlocks.

Another concern of synchronous IPC mechanisms is the asymmetric-trust problem. In a client-server scenario where multiple clients are relying on the same server, a badly intended partition could jeopardize the correct functioning of their concurrent partitions. To avoid these type of scenarios, a timeout strategy could be implemented. However [HE16] tells us that the timeout method is useless for DoS attacks, either because of wrong user implementation, or simply because there is no scientific or measurable way of determining an appropriate timeout value, which usually leads to timeout values of either zero or infinite.

- **Asynchronous communication** or non-blocking communication or event-based communication: Contrary to synchronous communication, in asynchronous communication the partitions do not meet at a specific moment, thus eliminating blockage in any of the partitions. This method of communication is usually paired with a separate event mechanism to notify the receiving partition that there is a new message to be received [RN93]. This separation between data and event channels allows the partitions to be executing while waiting for a message or a message response. However, this type of policy can induce kernel buffering and consequently, an additional data copy.

  When paired with an event mechanism however, which usually is the case for a complete IPC mechanism, this asynchronous communication method is still prone to DoS attacks, as one partition can flood the other with message events, preventing its correct functioning [LIJ97].

Concerning the IPC mechanism's data channel, these can be divided into:

- **Shared Memory**: This method consists on having a block of allocated memory which multiple partitions on the system can have access to, as a way of serving as the data channel for the IPC mechanism. It is a more efficient but less secure policy, as it needs to allocate a block of memory accessible by more than a single partition, making it an easy target for buffer-overflow

attacks [GKS94]. A bunch of strategies could be implemented to circumvent such problem, mainly with an access control strategy supported by a kind of execution privilege level.

Although most systems implement their own design of a shared memory IPC mechanism, more and more solutions are using the VirtIO technology standard. In [PGLA15, OMC+18] are shown different ways of using the VirtIO interface, which will be explained later in further detail.

- **Direct Transfer**: Copies data directly between send and receive buffers, thus reducing an extra copy from the shared memory approach [DPNJ08]. This entails that a dedicated kernel module [JSCP05] is responsible for message passing between the system's partitions, incurring necessary system calls and overall overhead.

Finally, all different approaches need to take into account the IPC destination. More traditional IPC mechanisms, e.g. the original L4 mechanism, used threads as destinations for IPC operations. Thus, a malicious entity could gather information from the system, e.g., to create an effective attack [EH13, LIJ97]. To counter such threat, the concept of endpoint was introduced, providing an abstraction layer for the communication mechanism where messages can be read or written.

In further sections, this document will present more in-depth expositions of more IPC standards and specific IPC mechanisms of established OSes and hypervisors that relate to this thesis' work.

2.1.4.2 Hardware-assisted IPC

In recent years, there has been a surge of hardware-accelerated IPC mechanisms. These usually consist of modified, standard communication mechanisms with hardware accelerators for data transfers. These mechanisms have emerged from the need of improving IPC performance and security without increasing software overhead. As mentioned before, the IPC mechanism is of utter importance in a critical embedded system, and having it migrated to hardware, even if only a small percentage of the whole mechanism, can greatly decrease communication overhead and help guaranteeing time-critical tasks.

Some solutions implement DMA-based (Direct Memory Access) memory copies, as it can access memory independently of the CPU, and thus accelerate these transactions. Projects such as [YBYW10, DPNJ08] make use of DMA to aid in communication performance. In [YBYW10, ABYY10] we learnt that there may

be some inherit problems on making use of DMA-based transfers for IPC mechanisms for virtualized systems. The first problem mentioned in [ABYY10] is the intentional or unintentional access to memory regions that an untrusted device is not allowed to access. Secondly, DMA might be unsuitable for use in virtualization environments by guest partitions, since these guests, do not usually know host-physical addresses and the device is unaware of guest-physical address spaces. Lastly, some legacy devices do not support long addresses and therefore cannot access the entire physical memory. [YBYW10] shows us how an I/O Memory Management Unit (IOMMU) can be used to solve most of these problems, but also the subsequent overhead caused by mapping (and unmapping if done during runtime). It follows by suggesting a basic set of rules that any DMA mapping scheme must satisfy: the DMA device must have all guest physical addresses accessed mapped in its IOMMU translation table; every mapping in a given device's IOMMU translation table must have a corresponding mapping in the CPU MMU translation table for the Guest using that device; every DMA-accessed address must me mapped both as guest-physical address and corresponding host-physical address. Furthermore, [SG12] proves that it is possible to make use of DMA-based memory transfers in a virtualized environment without making use of additional hardware, i.e. a dedicated IOMMU, only using a standard MMU.

Other solutions however, implement dedicated hardware accelerators instead of using DMA-based communication strategies. Dedicated hardware accelerators in FPGA SoCs are not new to the computing world, being used as early as the year of 1963 [EBTB63]. Reconfigurable computing sits in the middle of the hardware-based Application Specific Integrated Circuit (ASIC) and software solutions, combining the much higher performance of ASICs with the flexibility of software [CH02]. From matrix multiplication [DFK+07] and matrix floating-point computation [KGG06], to solving of complex imaging algorithms [Zem02], hardware-programmable alternatives have been explored in a myriad of different ways. So, as long as FPGA-enabled SoCs were small enough to be used for embedded systems solutions, hardware-acceleration has been used alongside it. Regarding IPC mechanisms, hardware accelerators have been used to reduce overhead induced especially by the memory copies, although almost-complete hardware-based IPC mechanisms have been implemented. One of these implementations is the in-house Message Passing Management Unit (MPMU) [GGM+16].

However, using a dedicated hardware peripheral for IPC can also be a dangerous task, if not taken care of. The same problems of buffer overflow and memory

attacks can still happen if unregulated, and it is up to the system designer taking these problems into consideration and either make use of memory manager peripherals to arbitrate memory copies, or implement a reliable hardware-based IPC mechanism that follows the aforementioned rules.

In further chapters we will discuss how these problems are avoided or even solved, either by the underlying hardware and software platforms used for this project, or this project's work itself.

2.1.5 VirtIO

With the various possible Linux virtualization systems, and with all of them having major differences in driver features and optimizations, there was a dire need for a solution that brought all of them together while simplifying the driver implementation and maintenance process. To achieve a standard solution, VirtIO emerged as the answer, consisting on a series of Linux drivers which can be adapted for various different hypervisor implementations by using an abstraction layer [Rus08]. Although initially only intended for Linux, VirtIO was later added to bare-metal and RTOS Guests by the OpenAMP project [BR16].

VirtIO project set out to achieve three basic goals: eliminate third-party interference in the Linux Kernel; create a common ABI for general publication and use of buffers; use their own virtio_ring infrastructure and the Linux API for virtual I/O devices for device probing and configuration. Ultimately, the project achieved its desired goals, with VirtIO successfully representing an abstract layer that provides a set of front-end and back-end para-virtualization drivers in order to ease the complexity of emulating a given device [OMC+18].

The goal was to create a transport abstraction layer capable of bringing efficient transport mechanisms to all virtual devices. To achieve that, VirtIO drivers start by registering themselves with the device type, and optionally, with the vendor name. Then, these virtual devices can be configured with a few operations, the first of them being reading and writing the necessary feature bits. The device will look for feature bits of specific device types that correspond to features it wants to use. This feature bit acknowledgement is known by both the guest and the driver. The second operation, reading and writing the configuration space, consists on configuring a structure containing information from the virtual device, which can be both read and written by the guest. In the third possible configuration, VirtIO will set and get an 8 bit device status word which the guest uses to indicate the status of the device probing operations. Lastly, the device reset operation is able to reset the device, its configuration and all its status bits.

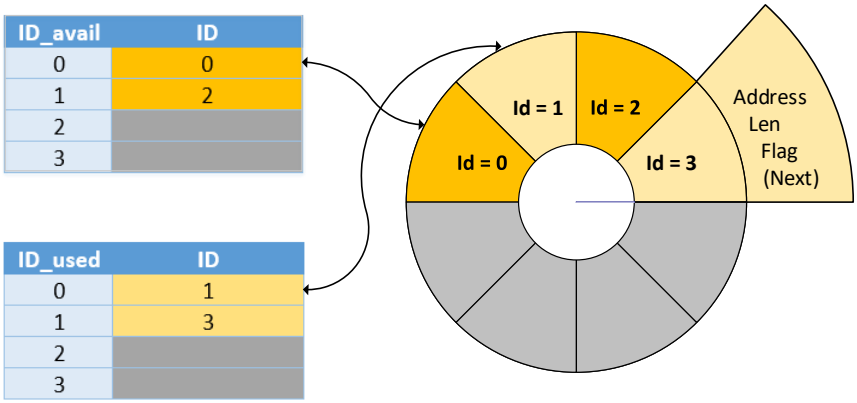


Figure 2.5: Virtqueue Circular Buffers

The most performance critical part of the API however, is the I/O mechanism itself, and not the configuration mechanism itself. As such, VirtIO implements virtqueues, created by a configuration operation (*find_vq*) when given the specific VirtIO device and the desired index number, being possible for a device to have multiple virtqueues (usually one for each input and output). This virtqueues are queues where guests write scatter-gather array buffers to be read by the host. The virtqueue operations structure consists of: an *add_buf* call to add a new buffer to the queue, which is complemented with the kick directive to notify the receiving end that a new buffer was added; a *get_buf* call to get a used buffer, returning the length of the buffer written by the counterpart; *enable_cb* and *disable_cb* (enable and disable callback) used to enable or disable the callback, which is equivalent to enabling or disabling a device interruption.

Regarding the transport layer itself, previous virtual device implementations were designed to leverage a multi-guest (specifically more than two) mechanism. To improve performance, VirtIO ditches this more versatile but slower I/O mechanism for an higher-throughput simpler mechanism, based on a standard method of high-speed I/O: ringbuffers [BDF+03]. VirtIO names its own mechanism as *virtio_ring*, consisting of: a descriptor array that can only be written by the master, containing buffer descriptions with length, address and additional flags; an "available" ring that also can only be written by the master, where the guest indicates what descriptors buffer arrays are ready to be used; an "used" ring which can only be written by the slave, where the host indicates which buffer arrays are used and ready to be recycled [Rus08]. Figure 2.5 depicts how the three buffer arrays relate to each other, where the descriptor is represented as a circular buffer containing information about both the used and available arrays.

VirtIO has a myriad of currently implemented drivers in many different device categories. One of these devices is the RPMsg device, a VirtIO-based messaging

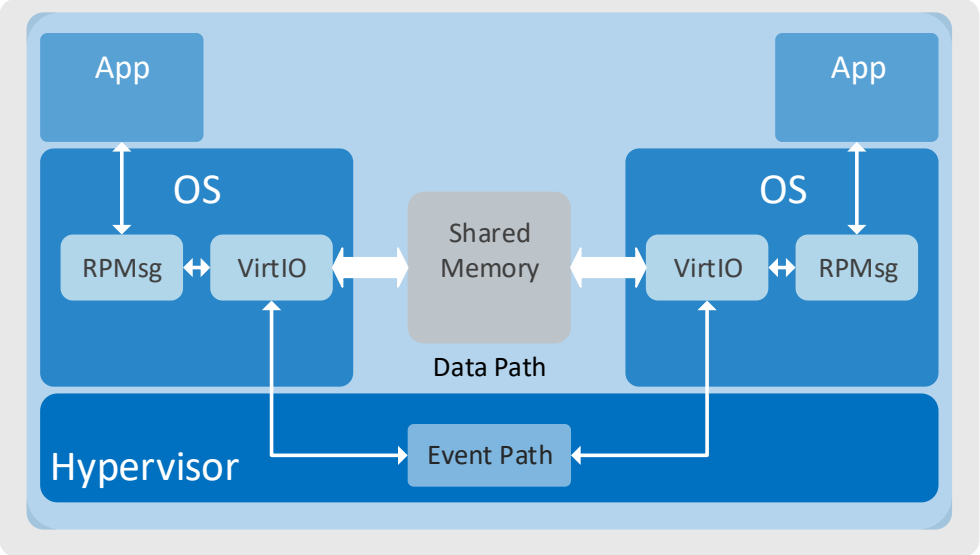


Figure 2.6: Virtualization Use-Case for RPMsg as IPC

bus that allows kernel drivers to communicate with remote processors available on the system. As such, RPMsg was naturally used for Inter-Core Communication strategies, but later adapted to be used as the transport layer of an Inter-VM communication mechanism. The normally back-end and front-end drivers are converted into master and slave communication drivers, respectively [OMC+18]. Although VirtIO enables bidirectional transport layers, RPMsg creates two separate unidirectional channels to achieve the bidirectional communication, as a way of better isolating communication. This way, it is possible for guests to enable one-sided notifications and be notified for the operations that they wish to be notified. This allows for both sides of the communication RPMsg channel to start the communication, as long as the master previously made the necessary configurations.

An IPC mechanism using the RPMsg communication protocol constitutes an asynchronous, shared-memory communication mechanism comprised of RPMsg Channels with the possibility of having multiple RPMsg Endpoints, containing their own address and callback routine. Also, being an asynchronous communication mechanism, it needs a separate event channel for notifications. Figure 2.6 represents a general use-case for using VirtIO and RPMsg as Inter-VM Communication, where both OSes can act as both master and slave to send and receive messages. Usually, the hypervisor will handle the event channel, notifying the guests as needed.

2.2 Related Work

As mentioned before, systems usually employ many different IPC implementations depending on the desired use and their own characteristics. Here are represented some essential for this project, either by employing similar features or by being important for overall project scope.

2.2.1 L4 Microkernel's IPC

First, we start by presenting an IPC mechanism implementation that is not too similar to the project at hands, but one whose generic implementation principles are followed by our mechanism. This implementation is the L4 microkernel and its IPC mechanism, as well as some of its later adaptations.

The original L4 microkernel, evolved from an earlier system, the L3 microkernel [Lie93a], which featured slow IPC performance with overhead costs on the order of the one-hundred microseconds [HE16]. [Lie93b] tries to tackle the issue of IPC for microkernel designs that severely lacked in performance and consequently overall system efficiency. Such issue that even led some to completely abandon the microkernel approach, since, as aforementioned, IPC is a fundamental component of any system, and an even more crucial one in microkernel designs. So, L4 follows an "IPC performance is the master" principle, keeping a strong focus on the performance of IPC operations, while still maintaining to the principle of minimal implementation. This principle, combined with the aim of generality, is still followed by each and every one of the modern L4 implementations.

L4 implemented a synchronous message-passing IPC mechanism to avoid buffering in the kernel and the management and copying cost associated with it [EH13]. This method works just fine in the original L4 implementation and on single-core systems of that time, since combining the context switch with communication minimises overheads. However, with the introduction of multi-core systems, a synchronous message passing mechanism became obsolete. As such, modern L4 versions added semaphore-like notifications to the original L4 IPC mechanism, either completely ditching the synchronous design (OKL4 implements virtual IRQs [HL10]) or maintaining the synchronous design and adding the notification mechanism. Regarding the message structure itself, there were not many modifications done, being the most notable the abandonment of "long" IPC messages, specially in small, resource-critical embedded systems. More prominent, is the introduction of port-like endpoints [KEH+09], a strategy introduced to replace the older "threads

as the targets of IPC operations" [HE16]. Further, less relevant, modifications include:

- Removal of IPC timeouts to be replaced by a single flag to configure as polling or blocking mechanisms. [Sha03]
- Abandonment of the original assembly code for the higher-level C code.
- Replacement of some physical message registers with virtual message registers.

2.2.2 MINIX's IPC

Similar to the IPC mechanism from the previous section, our implementation takes inspiration from the general principles of the MINIX's IPC system, rather than its design choices and structure.

Besides being fundamental in terms of performance and efficiency of most systems, the IPC mechanism must be dependable for the whole system to also be dependable, and that entails new concerns in the security and safety departments, as this means that such mechanism has to be aware of not only self-threats, bugs and other programming mishaps, but also threats from untrusted code on untrusted system partitions. In [TGB+08], the MINIX system makes their case on the need for dependability of the IPC mechanism on a modular system that relies indiscriminately on it, as untrusted application and drivers make use of the mechanism, and should not be able to affect the rest of the system.

MINIX defines an IPC threat as an "unintended IPC action that originates in a buggy component and that may disrupt the core operating system" and follows by identifying an assortment of IPC threats, divided into three classes:

- the IPC Subsystem itself: some exceptional IPC requests can potentially be used as attacks on the mechanism itself, either by invalid call parameters (invalid IPC primitive, message buffer, endpoints, etc.), or by the misuse of global resources.
- Message Delivery: regarding source and destination addresses (unauthorized access, wrong device, spoofing) or either the message content itself (oversized messages in dynamic payloads) .
- Group Interactions with untrusted processes: untrusted partitions can affect IPC flow control (scheduling problems and DoS attacks) and block other system partitions, even trusted, more privileged ones, if badly designed.

MINIX also presents an extended version of the Asymmetric Trust Model, including both untrusted clients and servers as possible threats.

To deal with the identified threats, the MINIX 3 IPC mechanism followed an asynchronous and non-blocking nature. MINIX also implements endpoints tied to a single IPC process. The IPC message itself is small and of fixed-length, with a fixed message header containing an endpoint and the message type. Furthermore, MINIX implements a set of restrictions to the IPC mechanism to better control the untrusted partitions that make use of IPC.

With these policies, adding a few extra cautions, MINIX's system is capable of being threat-resistant, self-reliable and dependable, which should be ideal for any safety critical system.

2.2.3 On-Chip Message Passing Mechanism for IPC

A good example of an hardware-based IPC mechanism is the in-house IPC mechanism presented in [GGM+16]. The Message Passing Management Unit (MPMU) is an all-hardware IPC mechanism deployed in an in-house SoC based on an ARM-compliant softcore processor.

This DMA-capable MPMU is connected as a memory-mapped peripheral. At system boot, the hypervisor configures the number of partitions, desired communication mode, and which memory addresses it reserved for MPMU use. Then, each partition configures the MPMU with the addresses of its input and output buffers within its own address space [GGM+16]. This prior configuration, enables a fast communication mechanism, as there is no need for additional information exchanges during runtime. When a partition intends to send a message, it signals the MPMU, which, in turn, uses the pre-configured memory addresses to make a direct copy to the desired partition. Also, being hardware implemented, means that the hypervisor has no need of interfering in the IPC transfer. Furthermore, this mechanism is user-transparent and, from the guest's view, the only performed action was a write to a memory location. There are some trade-offs however, for this hardware implementation, such as an added hardware and memory costs in trade for the better latency in communication, as well as a fixed-buffer message size in trade for less software overhead.

Our implementation takes inspiration on the MPMU hardware IPC mechanism for its performance benefits, specially data throughput and latency decrease, but will not implement the whole of the mechanism in reconfigurable hardware. Instead, the more performance critical tasks will be offloaded to hardware while maintaining some software configurations. More on this will be explained further.

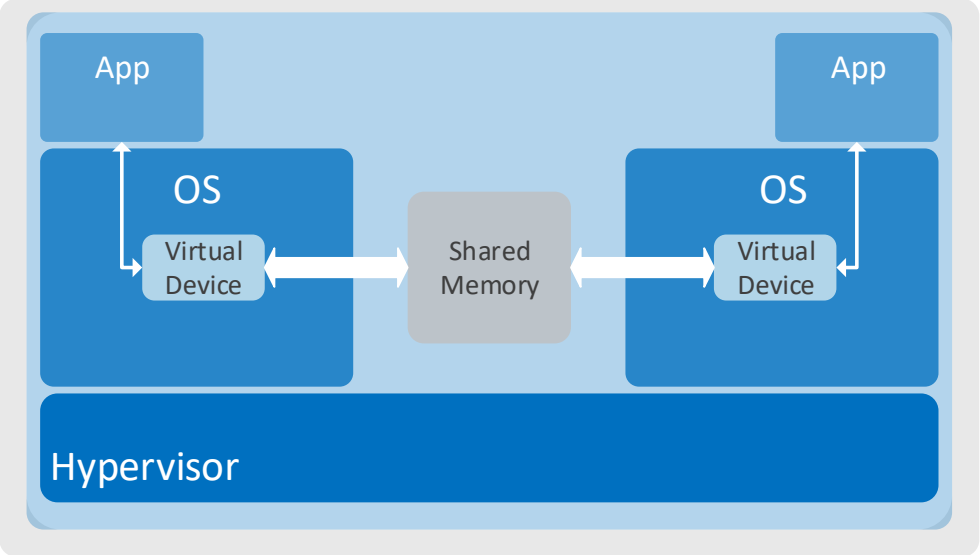


Figure 2.7: Use-Case for Virtual Devices as IPC

2.2.4 Virtual Device-based IPC

The following section shortly describes a series of hypervisors that make use of device I/O virtualization strategies to implement shared memory IPC mechanisms.

Figure 2.7 represents the general use-case when using a shared-memory approach with the aid of virtual I/O devices as interface. Many implementations also include a separate event path, either in the hypervisor layer, or as another I/O device.

2.2.4.1 Xen's IPC

The Xen hypervisor, presented in [BDF+03], is a multi-guest OS hypervisor capable of hosting up to one-hundred virtual machines. This hypervisor has been explored extensively and is even featured in aviation applications, after being modified to follow the ARINC 653 standard [CI06]. This standard consists on a partitioning method that reduces development costs, reduces system weight, and lowers certification costs by consolidating several applications on one computing resource, yet keeping them isolated [Van10, LLK08].

The main control mechanism for the high-level issues of the Xen hypervisor is the Domain 0 partition. One of the many tasks of this domain 0 is to handle device I/O virtualization. It is also through this mechanism that Xen implements its IPC communication strategy. Like RPMsg in VirtIO, Xen implements unidirectional circular ring buffers via shared memory mechanisms at kernel level, through the

Dom 0 partition. This communication is used for accessing hardware devices, as well as communication between Guest VMs.

2.2.4.2 SASP's IPC

SASP, or Secure Automotive Software Platform, is a project that implements a dual-OS TrustZone-based hypervisor (V-Monitor) with a specialized TrustZone-assisted device access mechanism. This mechanism restricts the GPOS from directly accessing system devices.

V-Monitor makes use of this mechanism to implement shared-memory IPC, starting by having the GPOS in the normal world making a device access request, followed by the context switch to the secure world after V-Monitor is notified, which will then copy the desired data.

2.2.4.3 SafeG's IPC

SafeG, or Safety Gate [SHT10], is a dual-OS TrustZone-assisted hypervisor supporting integrated scheduling, strategy which, SafeG claims, enhances the responsiveness of the GPOS while not affecting the determinism of the RTOS. SafeG's biggest claim is its health monitoring mechanism present on the RTOS, SafeG monitor, capable of suspending, resuming and restarting the operation of the GPOS. This is possible due to TrustZone's design: the secure world can access and has more privilege than the non-secure world.

As for device sharing, SafeG follows a re-partitioning approach which consists on "dynamically modifying the assignment of devices to each OS at run time" [SHT12]. This approach reduces overhead as it allows both OSes accessing the devices directly. Two different re-partitioning mechanisms are implemented in SafeG: pure re-partitioning, consisting on the dynamical re-partition of devices between the guest OSes, similar to *hotplugging*; hybrid re-partitioning, eliminating the need of device resetting at each device partition switch, as some of the configurations will be maintained throughout the system execution, and others, the necessary ones, will change when switched between secure and non-secure partitions.

As for its communication mechanism, *dualoscom* [SHT13], SafeG implements a shared-memory communication mechanism at user-space level, but it allows for untrusted, privileged applications to manage this shared memory block, which poses as a safety concern, with potentially allowing untrusted, unprivileged applications from accessing this shared memory. Like VirtIO-based communication

mechanisms, *dualoscom* also makes use of virtual I/O, as well as dividing communications into data and event path. However, it does not implement a standard interface.

2.2.4.4 Rodosvisor's IPC

Rodosvisor [TDL+12] is an in-house ARINC 653 Quasi-compliant hypervisor supporting multiple VMs. As well as other aforementioned hypervisors, Rodosvisor implements virtual I/O devices. Also, it relies on this virtual I/O for inter-VM communication, with an inter-VM framework deployed as a part of the hypervisor's *CCommManager*.

The shared-memory communication mechanism features:

- Binding communications unidirectional channels.
- Hypercalls for: channel creation, mapping and unmapping; send and receive data; start and end execution.
- Slave VM monitoring.

3. Platform and Tools

This chapter will present the research platform and tools used for the development of this project. Some of these were project requirements, while others were carefully chosen to meet the criteria imposed by the project scope. It starts by presenting the Zynq SoC ZYBO board and why this was the chosen hardware platform, along with an in-depth look at the more relevant components. Following, is a detailed exposition on the hypervisor used in this project, the in-house LTZVisor, as well as the IPC mechanism later added to this hypervisor, the TZ-VirtIO.

3.1 Zynq Platform

To correctly make a decision on the hardware platform used in this project, we must first evaluate the requirements as to complete this project. The identified requirements were:

1. The chosen platform must be able to support the open-source LTZVisor software, more specifically:
    - include a Cortex-A series ARM processor;
    - the ARM processor must provide hardware-assisted virtualization facilities;
    - TrustZone-enabled ARM SoC.
2. The chosen platform must be able to run GPOSes.
3. The chosen platform must include integrated FPGA logic supporting TrustZone hardware extensions.
4. The chosen platform must include DMA-capable memory and peripherals supporting TrustZone technology.
5. The chosen platform must be cost-effective to achieve the desired goals.

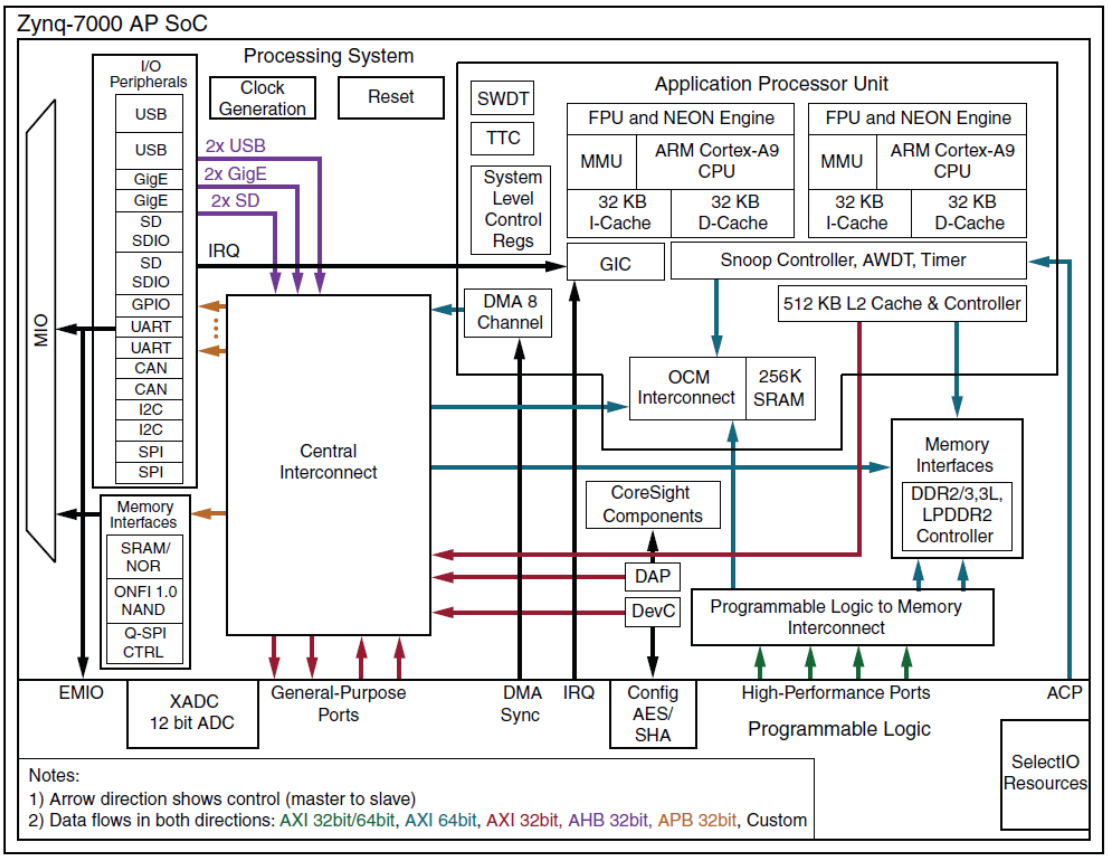


Figure 3.1: Zynq-7000 AP SoC Overview

To meet the first requirement, as well as its more specific constraints, we could check the LTZVisor's GitHub repository and verify the three currently supported boards by the LTZVisor project. These three SoCs, all from the same Zynq-7000 family, are all able to run the LTZVisor as it stands, providing hardware-assisted virtualization facilities and TrustZone security extensions. In fact, they meet all of the technical requirements that follow, as the ARMv7-A specification is complemented by an integrated MMU, and they all present FPGA and DMA capabilities. To finally reach a conclusion, we must choose the most cost-effective solution. This leaves us with the ZYBO SoC, the most affordable of the three options while still capable of achieving the goals set for this thesis.

The now retired ZYBO, produced by Digilent, is the smallest member of the Xilinx's Zynq-7000 family of SoCs based on the Xilinx All Programmable SoC architecture [Xil16, Figure 3.1]. It is divided into the SoCs Processing System (PS) side and the Programmable Logic (PL) side. The PS includes a dual-core ARM Cortex-A9 processor as the Application Processor Unit's (APU) main component. This processor includes 32 KB of instruction and data L1 caches per core and 512 KB of core-shareable L2 cache with a dedicated Snoop Control Unit (SCU) to

maintain cache coherency. The APU also features an Accelerator Coherency Port (ACP) to bridge the PS and PL sides, 256 KB of dual-ported on-chip SRAM (also referred to as OCM (On-Chip Memory)), an 8-channel DMA controller (4 channels for each SoC side) and a General Interrupt Controller (GIC). To interface with the 512 MB of out-of-chip DDR3 memory, a DDR Controller is included in the PS side. As for the rest of the I/O peripherals included in the PS, these include a GPIO, two up-to 1 Gigabit Ethernet controllers, two USB 2.0 Controllers, two SD controllers, two SPI controllers, two CAN Controllers, two UART Controllers and two I2C Controllers.

Finally, the PL side features a total of 28,000 logic cells, 240 KB of fast block RAM, 80 DSP slices and a Xilinx's own dual 12 bit Analog-to-Digital Converter.

To bridge both sides of the chip, there are a set of nine AXI interfaces including four 32-bit General Purpose AXI interfaces suitable for low-bandwidth PS-PL or PL-PS communications, four High-Performance AXI interfaces capable of burst transactions all mastered by the PL side, and an ACP interface between the PL and the SCU for cache coherency between APU and PL.

3.1.1 DMA

One of the most important hardware components for the development of this thesis is the DMA Controller (DMAC) present on the ZYBO. The DMAC uses a 64-bit AXI bus to transfer data between memory, from memory to peripherals and vice-versa, and between peripherals [Xil16, Figure 3.2]. To use the DMAC, the programmer stores the program code for the DMA engine, written in its own instruction set, to a region of system memory, which the DMAC will then access through its master AXI interface. This small instruction set, comprising of instructions for DMA transfers and management instructions to control the system, provides great flexibility for the system designer.

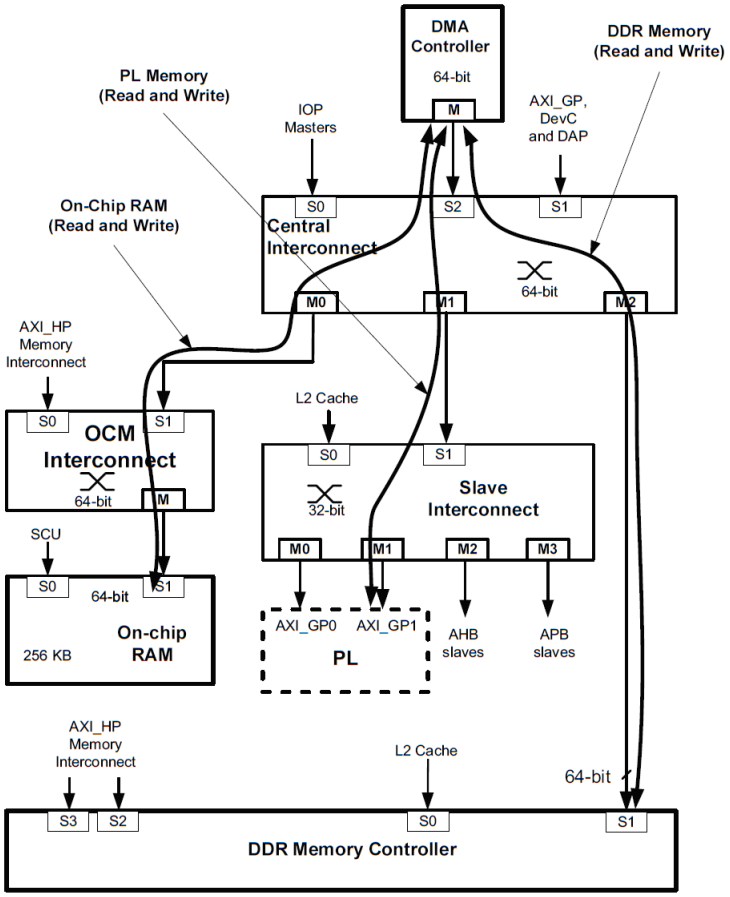


Figure 3.2: DMAC Channels

The controller is able to run eight simultaneous DMA channels, and as such, capable of moving big amounts of data without processor intervention. All of the memory and memory-mapped peripherals are within reach for this DMAC. Furthermore, the DMAC has two sets of control and status registers, one for the secure mode and another for the non-secure mode, thus guaranteeing physical separation between modes, as there can be no mixing between secure and non-secure created channels. To change the security status of the DMAC, the System-Level Control Registers (SLCR) should be used, which requires an hardware controller reset to take effect.

3.1.2 AXI

The Advanced eXtensible Interface, AXI for short, is the third generation of the Advanced Microcontroller Bus Architecture, AMBA3, an interconnect specification describing the connection and management of functional blocks in a SoC [ARM11]. AXI Full enables high-performance transferring of data between system's partitions, supporting burst transactions of up to 256 data transfers in each

burst, that allow write or read operations on sequentially addressed locations by only having issued the first address and the number of desired operations. There is also a subset of the AXI protocol, the AXI4-Lite specification, with a simpler control interface, but with much less throughput than its bigger sibling, only allowing one data transfer per transaction.

The AXI protocol defines a set of control and data transmission channels, which consist of: read address, write address, read data, write data and write response. Figure 3.3 demonstrates how the mechanics of an AXI Full read and write transactions work with visual representation of the aforementioned channels.

In a read operation, the master interface will, through the "read address channel", send the address it is trying to read along with other control signals, followed by the response of the slave interface, which should send the read data through the "read data channel".

On a write transaction, the master interface will send the address and control signals through the "write data channel" followed by the data to be written through the "write data channel". The slave interface will then respond through the "write response channel" sending a signal to signify the end of transaction.

This separation between channels enables simultaneous bidirectional transactions.

On the ZYBO, AXI is the central protocol used to connect every component on the board, as we can see in Figure 3.4. The physical AMBA Interconnect bus connects the various devices and peripherals using the AXI protocol, with a mix of AXI4-Full and AXI4-Lite protocols, depending on the needed data-throughput through the bus. In the PS side, these buses are used to connect the APU to the external peripherals, as well as to the DRAM. To connect between PL and PS sides, there are four General Purpose AXI ports, using AXI4-Lite protocol, and four High Performance AXI ports, using AXI4-Full protocol. There is also

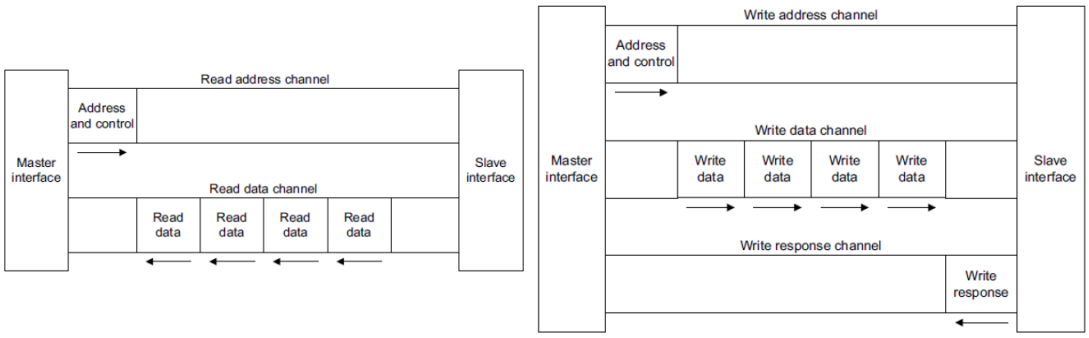


Figure 3.3: AXI Read/Write Transactions

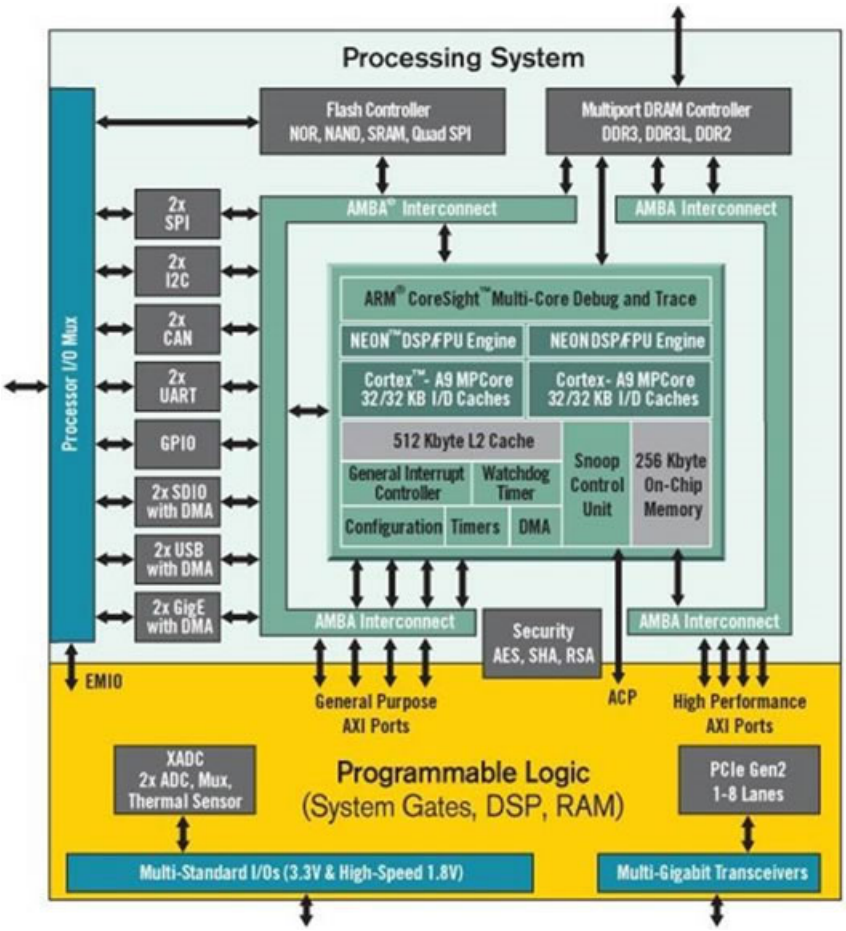


Figure 3.4: ZYBO SoC Overview

the possibility of using AXI Interconnect buses in the PL to connect between hardware modules.

3.2 LTZVisor

The software platform used in this project was the in-house open-source TrustZone-assisted Hypervisor, the LTZVisor, with its later implemented dual-OS VirtIO-based IPC mechanism, the TZ-VirtIO. This section describes both, and how they are relevant for the practical work of this thesis.

3.2.1 Overview

The LTZVisor [PPG+17], is an open-source lightweight TrustZone-assisted hypervisor created as a software platform to enable exploration of how TrustZone hardware extensions can be used in the aid of virtualization. To make the most

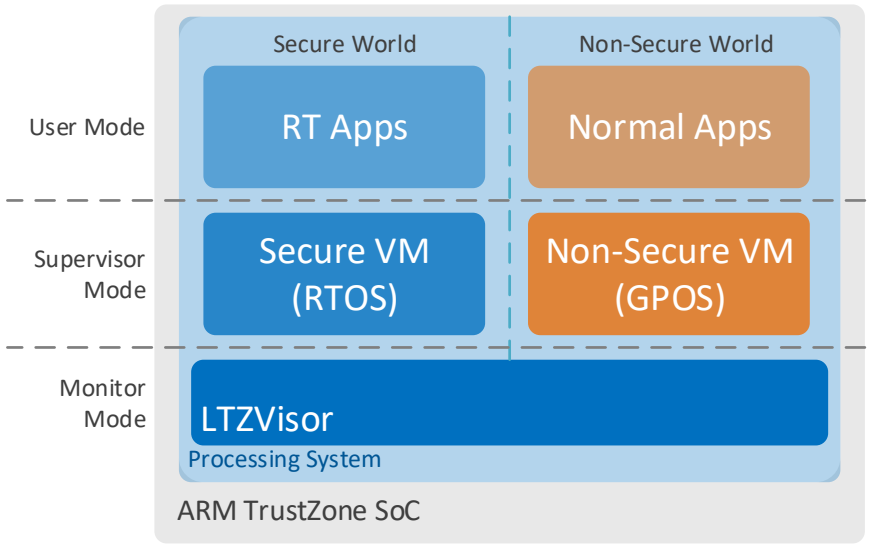


Figure 3.5: LTZVisor General Architecture

out of TrustZone hardware extensions as virtualization assistance, the LTZVisor follows three simple principles:

- principle of minimal implementation, relying on TrustZone technology for virtualization as much as possible, and diminishing the system's TCB, thus reducing the attack surface area;
- principle of least privilege, where resource access is given to components when absolutely necessary (TrustZone's separation by design helps greatly with this principle);
- principle of asymmetric scheduling, giving the critical secure side higher privilege of execution than the non-secure side ensures that timing requirements are met.

LTZVisor makes use of TrustZone's two virtual execution environments to host the privileged software in the secure world, while the non-secure world hosts the non-privileged software. The LTZVisor itself runs in monitor mode, the highest privileged processor mode available, where the processor state is always considered secure. In this state, the LTZVisor has full control over the system's resources and because of that, it is responsible for configuring the needed underlying devices for VM usage, as well as overall management of physical resources for the VMs. This includes managing the Virtual Machine Control Block (VMCB) of each VM at a partition switch, transferring the VM state previously saved in the VMCB to the

current processor context, and saving back to the VMCB the previous processor state.

Also running in secure mode is the secure VM which, unlike the monitor mode for the LTZVisor, is executing in a lower privileged level: the supervisor mode. Because of TrustZone's design, when executing in the secure state, the processor has the ability of viewing the non-secure side, and thus interfere with the non-secure VM. So, the OS running on the secure world will make part of the LTZVisor's TCB. Also, the fact that it is running at an higher privilege mode than the non-secure side, means that the secure side is the perfect candidate for hosting the RTOS, with the higher privilege level helping on meeting time constraints. Finally, the GPOS will be run in supervisor mode in the non-secure world. This VM is completely isolated from the secure world (secure VM and LTZVisor) and, because it is running in a high privilege mode but not on the secure world, it cannot access any component on the secure world side. Figure 3.5 shows the general architecture of the LTZVisor including the partition of respective worlds and modes.

3.2.2 Virtual CPU

By design, TrustZone technology divides each physical CPU core into two virtual CPUs, one for each execution world. Also, to help with reducing the number of registers to be saved and restored in each partition-switch, there is one physical register bank for each world, however different from each other. The secure side's VMCB is comprised of 16 banked registers, consisting on 13 General Purpose Registers, and three Control registers: Stack Pointer (SP), Linker Register (LR) and Saved Program Status Register (SPSR) for the System Mode. On the non-secure world, the VMCB is composed of 25 registers that also encompass 13 General Purpose Registers, but only one SP, one LR and one SPSR for each of the Supervisor, System, Abort and Undefined modes. Not included are the 5 higher General Purpose Registers as well as the Stack Pointer, Linker Register and Saved Program Status Register for both the IRQ and FIQ modes, as these are world exclusive. Furthermore, the Monitor mode is, by design, uniquely dedicated to the secure world side. Almost all of the coprocessor registers are banked, only the System Control Register (SCTLR) and the Auxiliary Control Register (ACTLR) need to be preserved.

Regarding the unbanked registers, and because TrustZone's design does not allow for the non-secure world to read them, can only be read but not written when the processor is in the non-secure state, as every attempt to write them

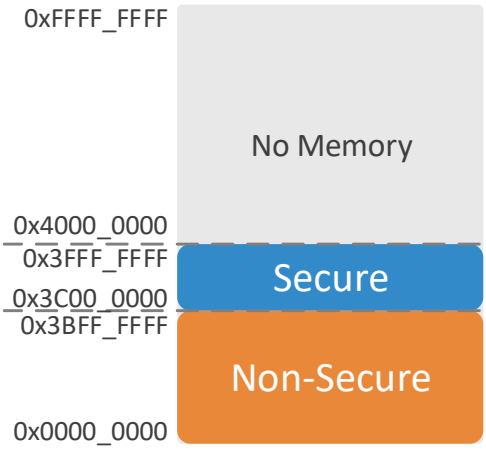


Figure 3.6: LTZVisor Memory Configuration for the ZC702 Board

will be ignored. This causes a problem for the non-secure OS, as some of these registers are needed for some hardware component setup, thus leading to a block in the non-secure GPOS boot process when these registers are attempted to be modified. To avoid this, LTZVisor fills some registers of the non-secure VMCB with the needed initialization values.

3.2.3 Scheduler

Virtualizing a real-time environment can be problematic, especially when an hypervisor schedules virtual CPUs, while a guest RTOS running over the virtual CPU schedules its own time-critical tasks. This causes an hierarchical scheduling problem, since ensuring that real-time tasks running on a virtual CPU and that both schedulers are designed to make sure the time constraints are met, requires a complex hierarchical scheduling analysis.

To avoid this problem, LTZVisor implements an asymmetric scheduling policy, thus guaranteeing that the non-secure GPOS is only scheduled during the idle periods of the secure RTOS. This way, the RTOS maintains an higher scheduling priority than the GPOS and is in fact the software component in charge of scheduling the VMs. This however, does not leave full control of the software stack to the secure RTOS, as the LTZVisor still runs at an higher privilege level.

3.2.4 Memory Partition

As aforementioned, TrustZone-enabled SoCs only provide MMU support for single-level address translation, and because traditional hardware-assisted virtualization relies on support for two-level address translation, a different strategy had

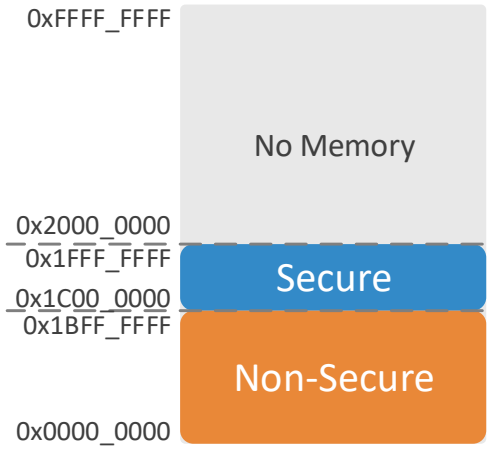


Figure 3.7: LTZVisor Memory Configuration for the ZYBO Board

to be used for mapping the guests virtual memory to host physical memory. As such, LTZVisor makes use of the TZASC to guarantee spatial isolation between the non-secure guest and the secure one. TZASC enables the partition of memory into different segments, and then allows setting those partitions to be configured as secure or non-secure. This does not allow for the non-secure world to access memory partitions configured as secure and, if the non-secure OS tries to access a secure memory address, an exception will be triggered and execution control will be redirected to the hypervisor.

Naturally, LTZVisor configures a bigger portion of the available memory as non-secure for the GPOS, as its memory footprint is much bigger than the combined secure RTOS and the hypervisor. Because memory regions can be configured with a granularity of 64MB, LTZVisor originally configures 960MB of memory, divided into fifteen 64MB memory segments (from 0x00000000 0x3BFF_FFFF). The remaining 64MB of memory are configured as secure for the hypervisor and the secure VM (from 0x3C00_0000 0x3FFF_FFFF). Figure 3.6 depicts memory configuration and partitions on the ZC702 board.

The original LTZVisor implementation however, was tested on the Xilinx ZC702 board which was endowed with 1GB of DRAM. As the ZYBO only has 512MB of DRAM, some alterations to the original ZC702 configuration were needed. Because memory segment granularity is of 64MB, the size of the secure partition could not be altered and the only remaining solution was reducing the size of the non-secure partition. The final configuration consisted on 448MB (from 0x00000000 0x1BFF_FFFF) for the non-secure partition and the remaining 64MB (from 0x1C00_0000 0x1FFF_FFFF) for the secure partition (Figure 3.7).

3.2.5 MMU and Cache

The MMU present on the Zynq-7000 SoC is TrustZone-aware, and although it does not allow for two-level address translation, it does enable each world to have its own set of virtual-to-physical memory address translation tables. This method reduces overhead at the time of a partition-switch, as translation lookaside buffer entries do not need to be invalidated.

As aforementioned, at cache level, the same world separation still applies. However, the NS bit is set by hardware and not accessible by system software. This means that there is no need for a cache flush at the time of a world-switch, greatly improving the LTZVisor's performance, since no cache operation needs to be performed by the software and cache management is handled by TrustZone's design. Furthermore, and because any attempt to enable or disable the L2 cache from the non-secure world side will be ignored (again, due to TrustZone's design), LTZVisor needs to enable both L1 and L2 caches before the non-secure GPOS starts booting.

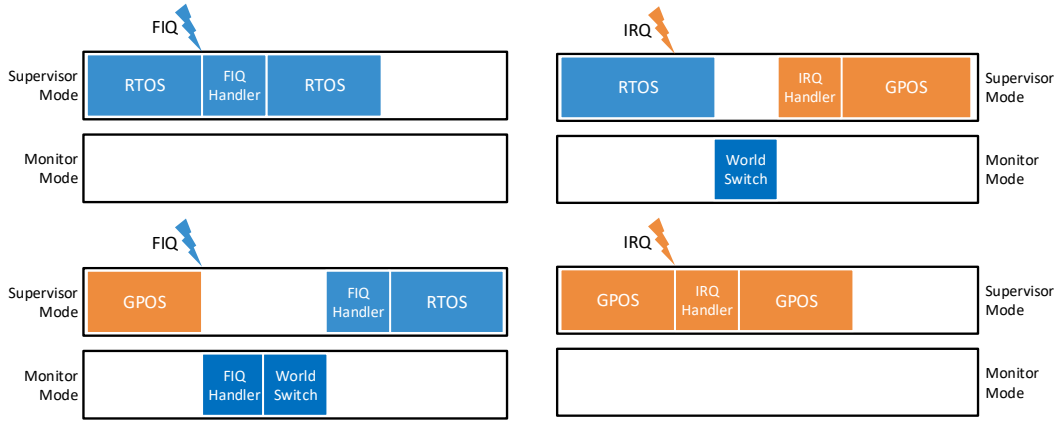
3.2.6 Devices

TrustZone technologies also allows devices to be configured as secure or non-secure, thus extending world isolation to the device-level. LTZVisor makes use of this feature by configuring the devices during boot time to the desired security world, and then implementing a pass-through policy, where devices can be accessed directly by the VMs. The devices will be configured as secure to be only used by the RTOS or and exceptionally by the hypervisor itself and so, triggering an exception handled by the hypervisor anytime the GPOS tries to access these devices. The devices that are intended to be used by the non-secure GPOS should, during boot time, be configured as non-secure, thus allowing for the non-secure world to access them.

3.2.7 Interrupt Management

TrustZone-enabled SoCs allow for both secure and non-secure interrupt sources to coexist, and also support for secure interrupts to be configured with an higher priority level than that of non-secure interrupts. Furthermore, both IRQs and FIQs can be configured to secure or non-secure interrupt sources.

LTZVisor, by design, configures interrupts coming from secure devices as FIQs and non-secure interrupts as IRQs. This is possible because the on-board GIC is TrustZone-aware and allows for all interrupts to be individually defined as secure



(a) FIQ Handling **(b)** IRQ Handling

Figure 3.8: Interrupt Handling in LTZVisor

or non-secure, through the Interrupt Security Registers set (ICDISRn). To be able to use the processor's FIQ mechanism, the FIQen bit in the CPU Interface Control Register (ICPICR) must be set. Furthermore, LTZVisor configures FIQs to be redirected directly to the RTOS without hypervisor assistance (even if the GPOS is currently executing), thus guaranteeing that no overhead is added to the interrupt latency of the secure guest OS. IRQs on the other hand, configured to be handled by the GPOS, do not affect the normal functioning of the RTOS, even if the interrupt happens when the secure VM is currently executing. Instead, the IRQ will be processed and handled as soon as the non-secure guest becomes active. Moreover, setting FIQs and IRQs to be handled by the secure world and non-secure world, respectively, prevents DoS attacks from the GPOS against the secure world. Figure 3.8 depicts LTZVisor's interrupt handling strategies for the various possible scenarios. In Figure 3.8a is shown how FIQ's are handled in the cases where RTOS and GPOS are executing. Figure 3.8b also depicts how LTZVisor handles interrupt for both running OSes, but this time for an IRQ.

3.2.8 TZ-VirtIO

Not included in the open-source version of the LTZVisor is the TZ-VirtIO, a VirtIO-based IPC mechanism. By using this VirtIO-based mechanism, LTZVisor intends to use a standard solution for Inter-VM communication, thus making use of later RPMsg adaptations to follow the latest trends in virtualization strategies, but modified for this open-source TrustZone-based hypervisor.

TZ-VirtIO implements an adaptation of the RPMsg API provided by Texas Instrument and OpenAMP for Linux and FreeRTOS VMs respectively. In Figure

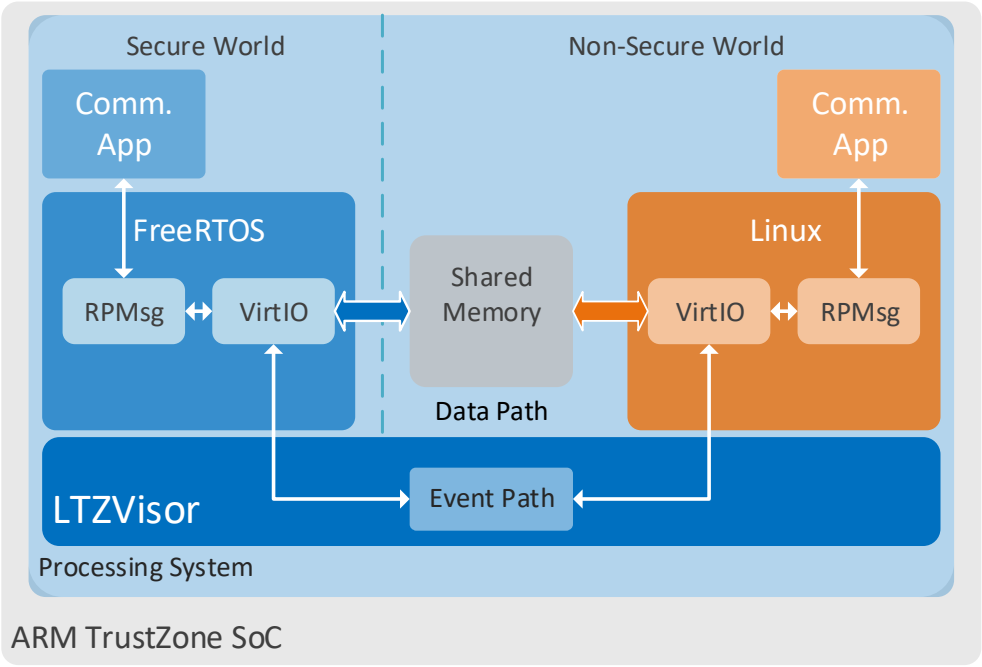


Figure 3.9: TZ-VirtIO Generic Architecture

3.9 is depicted the general architecture of the TZ-VirtIO IPC mechanism in the LTZVisor, including both RPMsg virtual devices and its VirtIO abstraction, as well as the data and event paths, and the user-space communication apps.

As aforementioned, this kind of IPC mechanism is of an asynchronous, shared-memory nature, and thus, it makes use of a non-secure block of memory for the transport layer (data path). This non-secure memory block is configured at compile time and statically allocated at boot-time. It is configured by the TZASC as non-secure so that both secure and non-secure worlds can access it. Although VirtIO allows for both the RTOS and GPOS to be the communication master, because the master is responsible for setting up the shared memory at boot-time and consequently the initial organisation of the VirtIO buffers, this task should fall on the more privileged secure OS to help insuring the isolation.

As for the notification mechanism, LTZVisor differs from other TrustZone-assisted trusted execution environment, since most of them implement a Remote Procedure Call (RPC) mechanism featuring SMC instructions for the notification events. However, this implementation does not suit the LTZVisor, as it requires an immediate world switch and consequent interrupt handling, which would not

be ideal, since the GPOS should not interfere with the secure RTOS to not jeopardise the real-time guarantees [OMC+18]. Instead, TZ-VirtIO makes use of Inter-Processor Interrupts (IPIs) to implement the inter-partition notifications of the RPMsg event path. However, because TrustZone technology blocks, by design, the non-secure world from interrupting the more privileged secure one, the direct triggering of an IPI was replaced by an interrupt request to the hypervisor via an SMC instruction, which forces an immediate switch to the monitor mode and subsequently entailing a slight increase in the partition-switching time. To circumvent this problem, the hypervisor will store the interrupt request in a circular buffer, which will be used to trigger the IPI to the respective OS during the next context-switch. This storing mechanism will prevent the GPOS from interrupting the time-critical RTOS.

Finally, because all of TZ-VirtIO resources are allocated statically, including the shared memory region and VirtIOs shared memory control data, this gives it the ability of being memory fault proof, thus preventing any unwanted access to out-of-bounds memory regions. This will reveal itself to be quite an useful feature later on.

4. hTZ-VirtIO: Hardware IPC Mechanism for LTZVisor

This chapter presents how the DMA IPC and the AXI IPC hardware modules were implemented, as well as the necessary LTZVisor and TZ-VirtIO modifications to integrate both the FreeRTOS and Linux Guests and the hardware components. It starts with an overview of the whole system and continues by delving deeper into each section.

4.1 Overview

Before a detailed description of each individual component over the next sections, this section will present the general hTZ-VirtIO system overview. Figure 4.1 depicts the overall architecture of the system, with LTZVisor and TZ-VirtIO as the software platform and the implemented AXI and DMA hardware modules in the board's Programmable Logic and Processing System, respectively.

The AXI module (hIPC) will be implemented in the SoC's Programmable Logic and connected to the Processing System via AXI protocol. As for the DMA module (DMA IPC), it will use the on-board DMA Controller for supervising the DMA data transfers. Furthermore, both mechanisms will access the TZ-VirtIO-allocated block of shared memory for reading and writing the IPC messages, thus being able to replace the TZ-VirtIO's software-only data path.

The TZ-VirtIO mechanism will stay mostly intact, with only some necessary changes, thus having the ability to choose between the three available methods for the data transfer mechanism: the previous TZ-VirtIO implemented software copy mechanism; software programmable DMA transfer; software programmable AXI transfer mechanism. The choice of the transport mechanism should be user-transparent and done for each individual case, as for each situation, each mechanism will bring their own advantages and disadvantages. To choose which mechanism to use with each data transfer, each resource will be extensively tested and

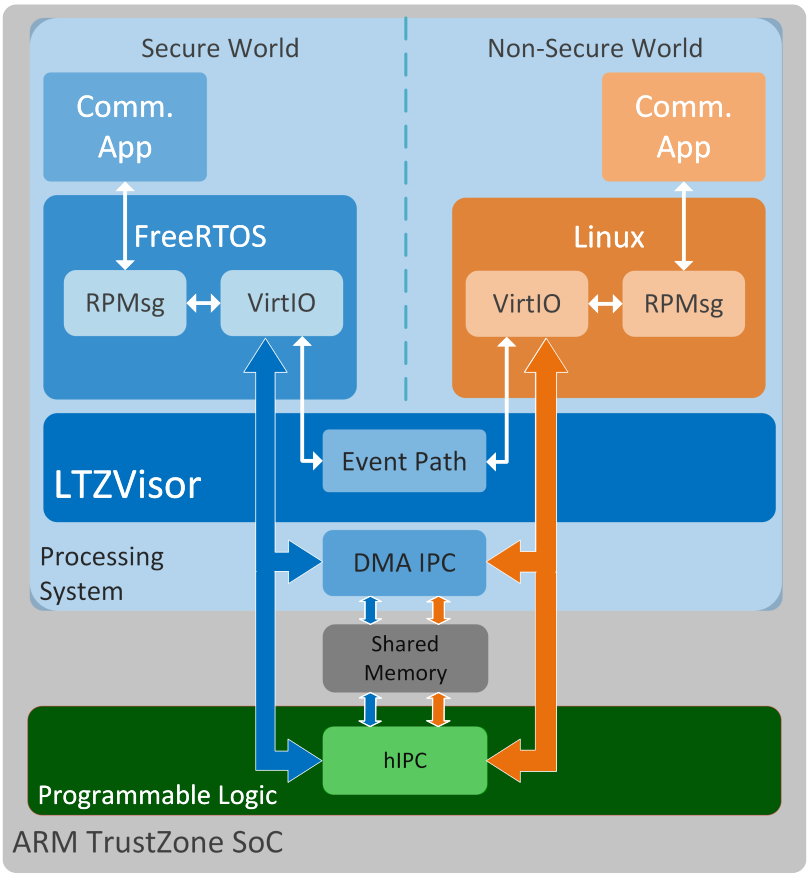


Figure 4.1: hTZ-VirtIO System Overview

their performance results will be taken into account. Moreover, each solution can also function individually, and thus, there is the option of only including one, or two, of the solutions to save hardware or software resources.

4.2 DMA IPC

The first implemented mechanism was the software-programmable DMA module. Using the DMAC of the Zynq-7000 was an obvious choice to consider when designing a system that intended to use hardware-based memory-to-memory transfer mechanisms. In this FPGA-capable SoC however, there were two available options for using DMA-capable transferring mechanisms: using the software-programmable DMAC available on the board's Processing System; use the hardware-programmable AXI DMA IP. Because the second option is more suitable for usage when the information is coming from an hardware module, and the other chosen implementation (AXI IPC) already uses this strategy, the choice made was to use the on-board DMA Controller.

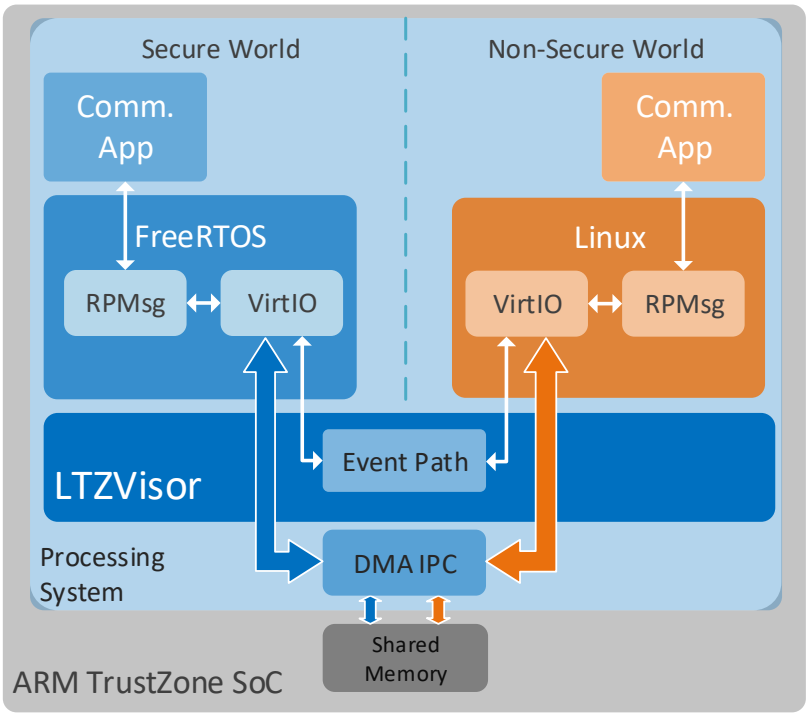


Figure 4.2: hTZ-VirtIO DMA-only System Overview

Furthermore, regarding the aforementioned problems of using a DMA-capable hardware peripheral for IPC in a virtualized embedded system, it can be clarified why this is not a problem for this implementation. First and foremost, because LTZVisor and TZ-VirtIO are implemented in a TrustZone-enabled SoC, the various hardware components can be set as Secure or Non-Secure. This means that both the block of shared memory used for IPC is configured as non-secure, as can be the DMA channels that are set from the Non-Secure world. This ensures that the untrusted DMA transfers cannot affect the secure partition. Figure 4.2 presents the hTZ-VirtIO System with only the DMA IPC implementation.

First off, because the DMAC is programmable and has its own instruction set, this implementation should start with the specification of the instruction set. The instruction format of the DMAC instruction set consists of a 8-bit opcode followed by a variable data payload of 0, 8, 16 or 32 bits, always prefixed by a "DMA" word to avoid confusion. The instructions can be for DMA channel usage only or for both DMA manager and DMA channel. Some of the instructions are common and similar to most instruction sets (DMAMOV, DMALD, DMAST), while others are DMA specific (DMAGO, DMAEND, DMAKILL).

To program the DMAC one needs to store the DMA program (a set of DMA

instructions) we want the DMAC to execute in its code memory. To easily program the DMAC, the needed C functions were created and added to the DMA IPC API. An example of a C function that adds a specific DMA instruction to the DMAC program is represented in Listing 4.1. All of these functions will receive as a parameter the pointer to the intended code memory address. The rest of the parameters will change depending on the instruction. In this case, the DMAMOV instruction needs the immediate value, as well as the directive to know to each of the three possible registers (Source Address Register, Destination Address Register and Channel Control Register) it should move the immediate value to. The function will return the value 6, as it is the number of bytes it will occupy in the code memory, which will be used in the API for pointer address incrementing.

```
1 int DMA_Instr_DMAMOV(char *DmaProg, unsigned Rd, uint32_t Imm){
2 /* DMAMOV encoding
3 *   47 ... 16   15 14 13 12 11 10  9  8 | 7 6 5 4 3 2 1 0
4 *   imm[32:0]    0  0  0  0  0 |rd[2:0] | 1 0 1 1 1 1 0 0
5 *
6 *   rd: 0 = SAR, 1 = CCR, 2 = DAR
7 */
8     *DmaProg = 0xBC;
9     *(DmaProg + 1) = Rd & 0x7;
10    *((uint32\_t *)(DmaProg + 2)) = Imm;
11
12    return 6;
13 }
```

Listing 4.1: DMAMOV Instruction

Other essential instruction pairs include the DMALD/DMAST, DMAGO/DMAEND, and DMALP/DMALPEND (DMA Loop and DMA Loop End instructions). A standard DMAC program comprised by these instructions will start by the configuration of the source and destination address registers, and the DMA channel configuration. Then, the DMALD and DMAST instructions will be contained by the DMALP and DMALPEND instructions, forming the DMA data transferring loop, where the DMALPEND instruction will check if the transfer is complete and, if not, jump back to the DMALP instruction address. Some remaining, or unaligned, bytes might be left out of the loop and transferred through consecutive DMALD and DMAST instructions, at the end of the loop. Then the DMAC should signal the end of the transfer. The program will be finished by the DMAEND instruction. Listing 4.2 presents a pseudocode DMA program.

```
DMAMOV Source Address Register;
DMAMOV Destination Address Register;
DMAMOV Channel Control Register;
DMALP
    DMALD;
    DMAST;
DMALPEND check if finished transaction;
DMASEV;
DMAEND;
```

Listing 4.2: DMA Program Pseudocode Example

4.2.1 Controller and Channel Configuration

Firstly though, the DMA Controller needs to be initialized and configured to handle the creation of DMA channels. Depending on the world security, the DMAC instance will have different physical base addresses and each must be accessed accordingly. Furthermore, if the DMAC is set in the secure mode it can only be accessed by the secure world, but it can create both secure and non-secure DMA channels. However, because there is a physical secure/non-secure separation, both are configured and set to be used by each of their respective worlds. This configuration is made at the LTZVisor's boot and only changed by the secure world, at most and only if needed, during a context switch operation. If set to non-secure, the DMAC still can only create and set non-secure DMA channels.

During execution, before starting a DMA transfer, the DMA channel to be used must be configured according to the expected transfer configurations. This also includes setting up the needed information for the later creation of the DMA transfer program, more specifically the information for the Channel Control Register, Source and Destination Address, and the transfer length. The available channel control parameters include the burst size and length for each source and destination, as well as if it is an address incremental burst. The security of the channel is also configured through the Protection Control bits. In the channel configuration stage, the source and destination addresses, as well as the transfer length, are also set.

Moreover, the setup of the channel interrupt for when the data transfer is finished should be done at this stage of the DMA channel configuration. However, because this is implementation specific, the details on how it is done will be further

explained in the section describing the integration with the LTZVisor. Nonetheless, at the most basic level, the DMAC should set the event for the corresponding DMA channel, which means tying together the platform's physical interrupt identification number (as each channel will have its own individual interrupt request) with the handler for each of the channels. In this implementation however, all of these interrupts are set to be handled the same way, as this implementation does not care about which channel executed the transfer.

4.2.2 Transfer Setup

After configuring both the DMA Controller and one of the DMA channels to use for the transfer, we are ready to then create the DMA program that will execute the data transfer. To make this process easier, a function that generates a DMA program was added to the API. This function will take all of the previously specified configurations for both the DMA Controller and channel and create a program with a similar structure to the pseudo-example presented in Listing 4.2.

So, after checking if the DMAC program memory is freed, the DMA program will be generated, starting by the 3 consecutive DMAMOV instructions that will configure the Source Address, Destination Address and Channel Control Registers.

The next step should be to program the data transferring loop, however, because the maximum number of burst transfers is 256 words, the procedure will be different if the transfer is longer than 1024 bytes, or 256 times 32-bit words. If the transfer is smaller than, or equal to, 256 words only a single loop will be needed to execute the whole of the transfer. So, the next instructions in the program will be: the signalling of the start of the transferring loop, including the transfer length (DMALP); DMALD and DMAST instruction pair; and ended by the DMALPEND instruction. On the other hand, if larger than 1024 bytes, the transfer will have to be done in more than one loop. To achieve this, the DMAC gives the ability to implement nested loops. An example on how to transfer 1024 words (4096 bytes) divided into four times 256 burst loops is presented in Listing 4.3.

```
DMALP
  DMALP
    DMALD;
    DMAST;
  DMALPEND 256 words;
DMALPEND 4 loops;
```

Listing 4.3: DMA Nested Loops

Another consideration that has to be taken into account is the possibility of tailwords and tailbytes. The first happen when the length of the transfer is not divisible by the configured burst length. For example, if the DMA channel's burst length is set to 4 words, and the transfer length is of 30 words, the DMAC should transfer 28 words in the main loop, and then transfer the 2 remaining words (tailwords) in another smaller, separate loop (example represented in Listing 4.4). Tailbytes happen when the desired transfer is not word aligned, meaning that the bulk of the transfer will be done in the regular way, and the unaligned tailbytes will have to be transferred separately, after reconfiguring the DMA channel to handle single bytes.

```
DMALP
  DMALD;
  DMAST;
DMALPEND 28 words;
DMAMOV Channel Control Register;
DMALP
  DMALD;
  DMAST;
DMALPEND 2 words;
```

Listing 4.4: DMA Tailwords Handling Example

After generating the transferring loop two other instructions remain: the DMASEV instruction, that will set the interrupt event for the specific channel (as aforementioned, the physical interrupt setup and handling will be later explored in the software platform modifications section); and the DMAEND instruction, meaning the end of the DMA program.

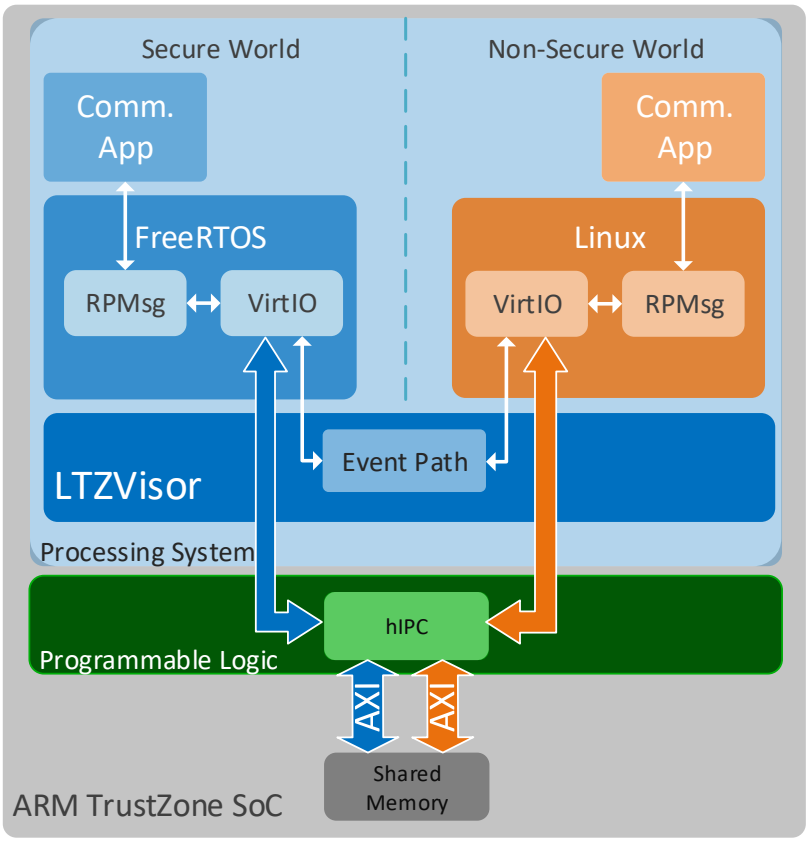


Figure 4.3: hTZ-VirtIO AXI-only System Overview

4.3 Hardware IPC

The other implemented hardware-based, data-transferring strategy was the AXI IPC. This implementation uses the FPGA, Programmable-Logic, section of the ZYBO board to achieve an application-specific hardware module capable of executing the best, tailored to the implementation, data transferring solution. Although the DMAC module present on the Zynq-7000 board already executes this same operation, providing a zero-copy (no CPU interference needed) hardware-based data-transferring mechanism, implementing our own hardware module should lead to both a performance boost and latency decrease. By implementing our own module, instead of using the on-board DMAC, we can follow a minimal approach, thus bringing unnecessary latency down. However, some drawbacks entail from this strategy, one of them being the obvious hardware cost brought on by the use of hardware resources. This value is also kept down by the minimal approach.

Like the DMA-based implementation, this hardware-based implementation

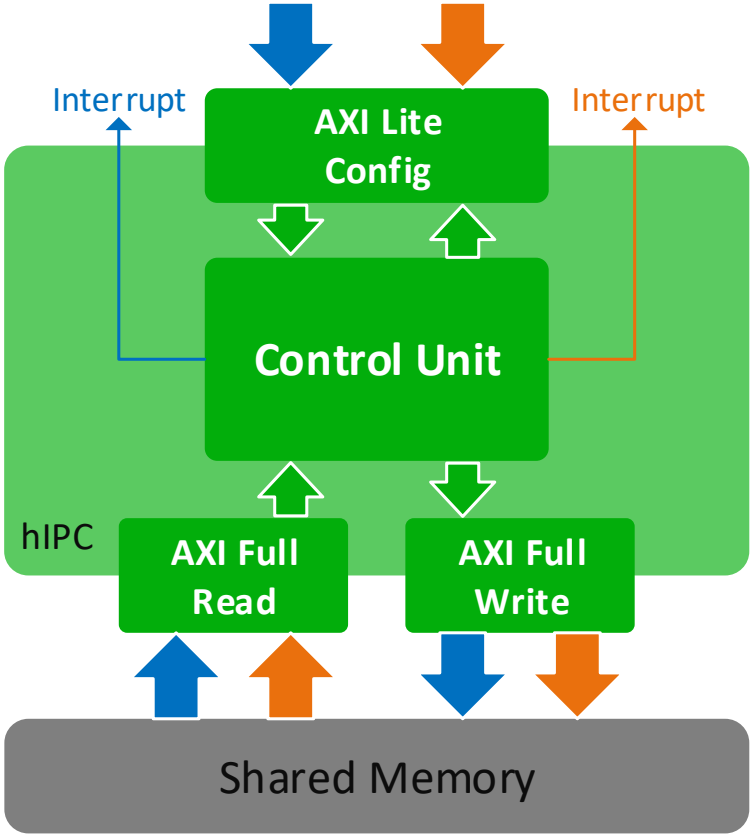


Figure 4.4: AXI IPC Module Architecture Overview

avoids the same problems of using hardware-based mechanism in a virtualized environment, in the same fashion. Just like the DMAC, the hardware module can be configured both as secure or non-secure, configuration which, when paired with the non-secure setting of the shared-memory block of the TZ-VirtIO system, ensures isolation between partitions, not allowing the secure world from accessing and modifying secure, trusted memory.

All of the aforementioned reasons lead to the choice of an hardware module as a theoretical viable option for the system. As such, the proposed hTZ-VirtIO system architecture with an AXI hardware module is represented in Figure 4.3. It consists on a modification of the TZ-VirtIO's data path, transferring it to an hardware-based solution which will receive the transfer configurations through an AXI Lite Port and execute the transfer via two, High-Performance, AXI Full Ports, one for read and for write. The event path is left almost untouched, with only an extra notification being added for the signalling of the end of the transfer.

4.3.1 Hardware Module Overview

As aforementioned, the hardware module itself will consist on a main Control Unit module that will manage the main execution flow of the IPC hardware module, as well as handling both Configuration and Transfer Ports, including the transferring of data between the Read and Write AXI ports. Figure 4.4 represents the general architecture overview off the AXI IPC module and the general direction of the data flow for both the configuration parameters, as well as the IPC message data itself.

At its most basic level, the hIPC module will receive the necessary configurations through the AXI Lite Configuration Port (Source and Destination Addresses, Message Length), as well as the trigger signal to start the message transfer, and using that information to execute a transfer between two memory blocks. With the given information, and immediately after receiving the trigger signal, the hIPC module will first trigger the read port, which will access the DRAM, more specifically the previously allocated Shared Memory portion, and read the desired memory addresses, starting at the configured Source Address and ending after reading the number of addresses that were set through the Message Length Configuration. In the meantime, the AXI write port will have also been triggered and started writing the data read by the other port in the address specified in the Destination Address Register and with the same length. After finishing the transfer, the control unit will trigger an interrupt, depending on the security of the transfer. Because these operations happen at the same time, some extra considerations have to be taken into account and they will be explained in more detail in further sections.

4.3.2 Configuration Port

To interface between the software and hardware layer, the hIPC module uses an AXI Lite port. By being register-based and memory-mapped, the AXI Lite port allows for the software layer to send information to the hardware module by simply writing directly to the previously allocated registers. In this case, the port is used as the configuration port, having available three configuration registers: Source Address Register, Destination Address Register and Message Length Register. Furthermore, to initiate the transfer after having set the registers, the software layer should write the trigger register.

Figure 4.5 represents the Configuration Port overview, with the four user-implemented registers which are then outputted from the port to the control unit

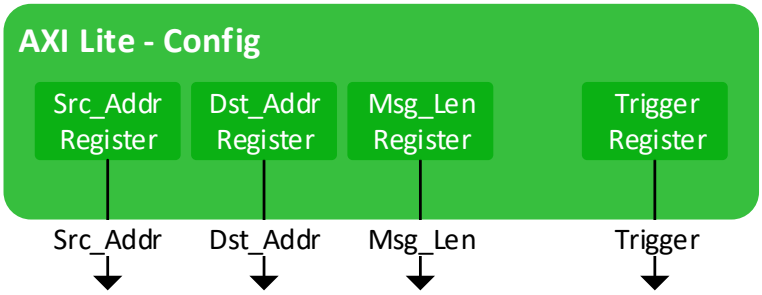


Figure 4.5: hIPC AXI Lite Configuration Port

module through wires. Not represented in the image are the control wires from the AXI protocol that execute the handshake between the interfaces.

4.3.3 Transfer Ports

The main mechanism of the data transferring operation is made up by the dual, unidirectional AXI Full interface unit. The AXI Full interfaces were used, instead of AXI Lite ports, as to achieve the highest possible throughput, since AXI Full has the option of using burst transactions up to 256 word-bursts, or 1024 bytes. Although a single AXI Full port is capable of performing both read and write transactions, this implementation uses two individual AXI ports. This choice, allied to the dual-port DRAM controller present on the ZYBO, makes this a much better choice performance-wise. This is because a single AXI port can execute both read and write operations but not simultaneously. Such operation is only possible by using two separate AXI ports. To keep hardware costs down, and because in this implementation these interfaces only need to work unidirectionally, this implementation follows a minimal approach, removing the unnecessary hardware resources that would enable the read port from writing in the memory and vice-versa. This not only saves on hardware costs, but also simplifies the overall implementation and execution flow.

There is a small drawback to this strategy however, which is the use of an extra physical AXI channel. So, this implementation occupies two of the high-performance AXI channels available on the board while still leaving two free channels for further use.

Figure 4.6 represents the overview of both of the AXI Full ports. Both receive information from the hIPC's control unit, namely, the address and size of the transfer, as well as the trigger signal to start the moving of data. Both also receive, from the DRAM controller the "done" signal, flagging the completion of the operation. The read data from the read port will flow towards the control

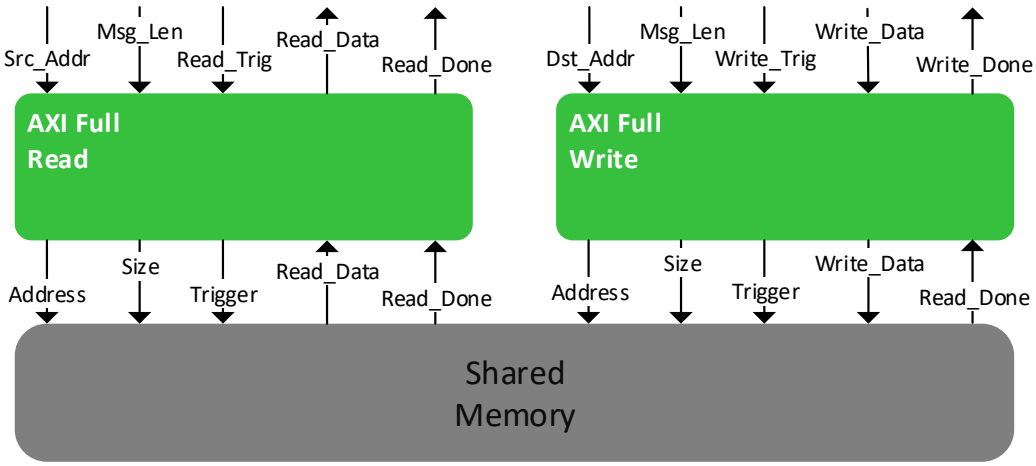


Figure 4.6: hIPC AXI Full Transfer Ports

unit module, which will then be handled and sent to the write port as quick as possible. However, some synchronization between the data coming from the read port and entering the write port is necessaryand it is executed by the control unit as it will be explained in the following section.

4.3.4 Control Unit

The last, and more crucial, section of the hIPC module is its control unit. This module is responsible for handling all the necessary configurations from, and to, the AXI ports, as well as the message data to be transferred. The module can be divided into four distinct sections. The first consists on the storing of the memory addresses for configuring the interfaces to read and write the messages to as well as the message length for configuring the length of the AXI burst. The control unit also manages the triggering of both of the AXI channels at their due time. The data handler sector manages the synchronization of the read and written data since the data is not stored, more than a few delay clocks, in the module, as is written directly to the final position. The final module, which can be included as part of the state machine portion, is the "Done Handler" module, which, when the transfer is finished, will trigger the hardware interrupt that will signal the software that the operation is complete. Figure 4.7 depicts the modules and inputs/outputs of the control unit module.

Because the AXI interconnect ports will not execute any operation until the trigger is set, the configurations registers are directly hardwired to the AXI wires that will be connecting to the DRAM Controller. This means that the Src_Addr register and Msg_Len register are connected to the address and size inputs of the

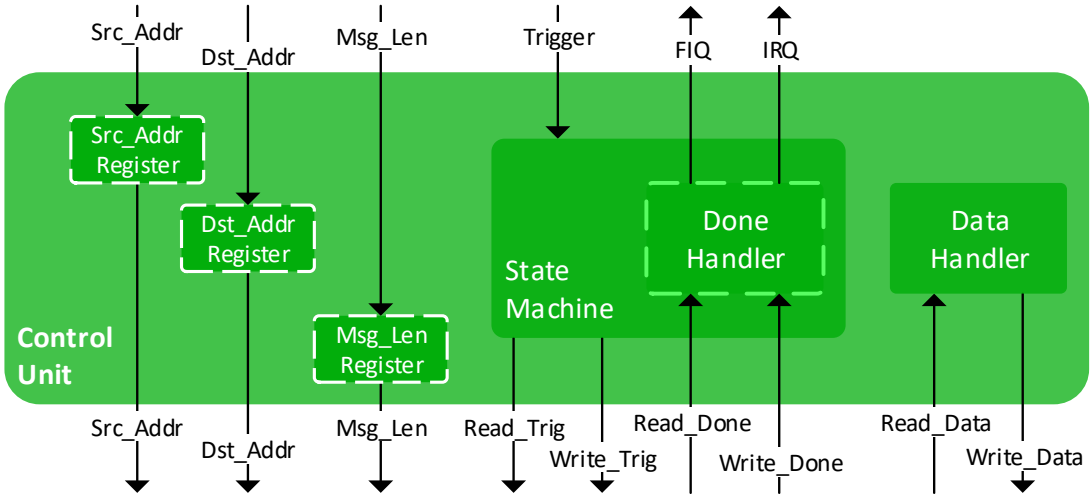


Figure 4.7: hIPC Control Unit

AXI read interconnect port, respectively, and the Dst_Addr register and Mes_Len register are connected to the address and size wires of the AXI write port, also respectively.

The next sector of the hIPC's control unit module is its state machine, responsible for handling the triggering of both read and write transfers and, through the *Done Handler* module, trigger both secure and non-secure interrupts. To better illustrate this behaviour, Figure 4.8 depicts the state diagram with the various states of the control unit module. When a trigger is received through the AXI Lite configuration port, the module will leave the idle state and enter the "Trig_Read" state, where it will trigger the AXI read interconnect to start the burst read transaction. Meanwhile, immediately after receiving the "i_rvalid" signal from the read channel, the module will move to the "Trig_Write" state, and consequently trigger the start of the write transaction. After triggering both transactions, the state machine will transit to the waiting state, and will remain there until both read and write transactions are completed and the control unit receives the "Read_Done" and "Write_Done" signals from both read and write interconnects, thus switching to the "Signal" state. In this state, the *Done Handler* module will check the security of the transfer and signal either a FIQ or IRQ, promptly switching back to the "Idle" state to wait for a new trigger.

Finally, the *Data Handler* module must synchronize the read data with the write data. This is because the reading and writing of data are done simultaneously, but the data to be written is a few clocks behind, and it needs to be written in the correct addresses. If the data is not on the correct position to be written,

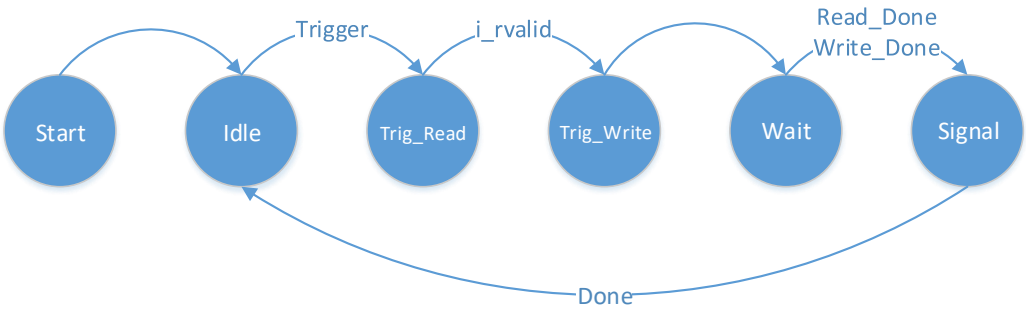


Figure 4.8: Hardware Module General State Machine

the address is selected and the AXI Port is ready to write, the wrong data will be written in that address. To avoid this, the data to be written is a few clocks delayed to reach the correct address. Furthermore, because both channels are accessing the memory at the same time, some extra delay cycles are needed, as the DRAM controller will not allow the memory to be written during a few cycles, but the read port keeps reading the data.

4.4 LTZVisor Integration

After implementing the individual memory-transferring components, the next step would be to integrate the individual components with the software platform. This includes porting the necessary OSes to the LTZVisor, the VirtIO-based communication mechanism between OSes and the final integration of the implemented mechanisms with the rest of the system. Before adding the guests OSes, the LTZVisor source file tree, as represented in Figure 4.9, must be slightly changed to run FreeRTOS and Linux.

If not configured to run any non-secure guest, the first step should be to set the needed memory blocks to non-secure, as well as setting as non-secure all of the peripherals that the non-secure guest, the Linux OS in this case, will need to access. This is done in the board initialization function of the LTZVisor, where all of the devices security is set through the various TrustZone configuration registers. To be able to run Linux, some modification need to be made to some of these registers.

Regarding the memory security, because one segment is enough for both FreeRTOS and the LTZVisor, the remaining seven segments are set to be non-secure through the TZ_DDR_RAM register. Linux also needs to use the board's global timer and for that the SCU access control register and SCU non-secure access

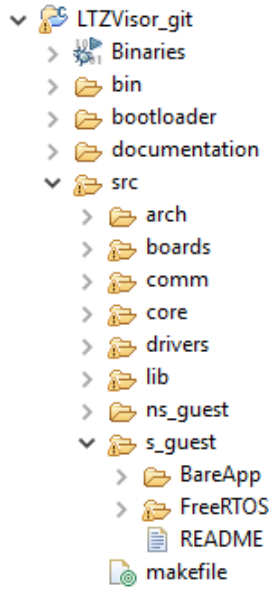


Figure 4.9: LTZVisor's File Tree

control register have to be set. Furthermore, for the non-secure world to access the AXI Lite memory-mapped peripherals, the TZ_FPGA_M and the security_fssw_s0 registers have to be set, thus enabling the NS signal propagation through the AXI general purpose ports and to the FPGA logic. Listing 4.5 represents the "board_init" function with the previously mentioned TrustZone security registers. To be able to modify these registers, they need first to be unlocked by writing the unlock key to the SLCR unlock register. After changing the desired registers, the lock key must me written to the SLCR lock register, thus locking these registers from modifications.

```
1 uint32_t board_init(void){
2 // Unlocking SLCR register
3 write32( (void *)SLCR_UNLOCK, SLCR_UNLOCK_KEY);
4 ...
5 // Handling DDR memory security (first 7 segments NS)
6 write32( (void *)TZ_DDR_RAM, 0x0000007f);
7 // SCU Access Control Register
8 write32( (void *)SCU_ACR, 0xF);
9 // SCU Non Secure Access Control Register
10 write32( (void *)SCU_NSACR, 0xFFF);
11 // M_AXI_GP0 Master Security
12 write32( (void *)TZ_FPGA_M, 0x3);
13 // M_AXI_GP0 Slave Security
14 write32( ( void *)security_fssw_s0, 0x1);
15
```

```
16 // Locking SLCR register
17 write32( (void *)SLCR_LOCK, SLCR_LOCK_KEY);
18 return TRUE;
19 }
```

Listing 4.5: Board Init Security Setting

Furthermore, to be able to use the global timer, the Linux OS needs to receive interrupts from the global timer peripheral and thus, these also need to be set as non-secure. This way, the interrupts are routed directly to the non-secure world with no Hypervisor assistance. To achieve this, during boot, LTZVisor configures the global timer interrupt as non-secure by setting the correct ICDISRx register as non-secure in the GIC. Moreover, although not crucial for the functioning of the Linux OS, the UART peripheral interrupt is also set to non-secure, so that is able to be used by both secure and non-secure OSes.

```
1 uint32_t ltzvisor_hw_init(void){
2 ...
3 // Linux/FreeRTOS UART Interrupt - IRQ 82
4 interrupt_security_config(UART_1_INTERRUPT, Int_NS);
5 // Global Timer Interrupt - IRQ 27
6 interrupt_security_config(GLOBAL_TIMER_INTERRUPT, Int_NS);
7 ...
8 }
```

Listing 4.6: Interrupt Security Setting

In Listing 4.6 the LTZVisor, in the hardware initialization function, sets the security of both the UART and the global timer as non-secure, using the previously defined "interrupt_security_config" function.

4.4.1 Guests

Because the open-source version of the LTZVisor project, available on GitHub [Pin18], only includes bare-metal guests, and the intended communication use-case involves a dual-OS configuration, more specifically an RTOS and a GPOS, these OSes need to be ported. The chosen OSes were: FreeRTOS for the secure, time-critical world, and Linux for the non-secure, general-purpose side.

LTZVisor's, by design, includes the secure guest in the hypervisor source code, compiling them together as a single image, but adds the non-secure guest pre-compiled image during the LTZVisor's boot sequence. This is due to the tight relationship between the secure OS and the hypervisor, specially because of the aforementioned hypervisor services that are transferred to the secure guest.

As such, modifications needed for both secure and non-secure guests are different from each other. FreeRTOS needed to be modified and included in the hypervisor's file tree, and compiled alongside it, while the Linux image needed to be modified and compiled individually, and later copied to the system.

Because this thesis' implementation is based on the TZ-VirtIO project and implementation, FreeRTOS and Linux versions were chosen according to previously tested versions for the sake of coherency and simplicity.

4.4.1.1 FreeRTOS

Regarding the secure world RTOS, the chosen OS was the FreeRTOS, more specifically version 7.0.2. Although this is an older version (more updated version as of the writing of this thesis is version 10.1.1), it serves the purpose for the system implementation.

First, the FreeRTOS source code has to be copied to the LTZVisor's file tree (Figure 4.10). Then, because both are compiled together in the same image, the LTZVisor's makefile needs to be altered to allow compiler and linker to add the FreeRTOS files during the making of the image. This includes adding the new FreeRTOS folder paths for both the source files and the included libraries, as well as adding the "objects.mk" files for the output object files.

Thee modifications to the FreeRTOS source code for the porting to the LTZVisor, mainly consisted on adding the needed FIQ handlers to FreeRTOS. This is due to two reasons: the secure side is set to handle all FIQs; the secure world is responsible for the partitions' scheduling, due to the LTZVisor's asymmetric scheduling policy. As such, two new situations have to be considered for FreeRTOS to handle as FIQs: the normal FreeRTOS scheduling mechanism has to be changed from IRQs to FIQs; FreeRTOS must deal with FIQs from the non-secure world, which will either be triggered by SMCs or the system tick.

4.4.1.2 Linux

The main difference between the secure and non-secure OSes included in LTZVisor is that, while FreeRTOS is compiled alongside the hypervisor layer, the

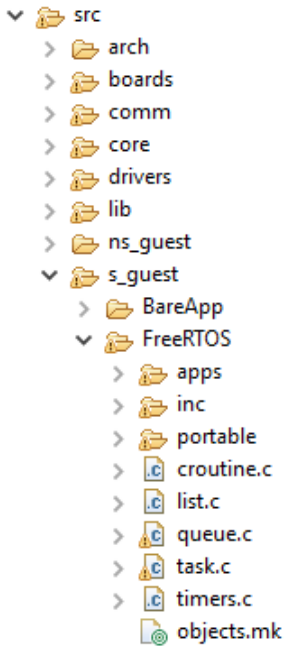


Figure 4.10: FreeRTOS Source Code in LTZVisor's File Tree

non-secure Linux OS is compiled by itself, and the image is then included during the LTZVisor's boot. This brings forward some considerations: the non-secure OS does not need as many modifications; this strategy is much closer to a full-virtualization strategy. Though, some modifications are still required, these mainly consists on modifying the ".dts" device tree file, disable FIQs, and compiling Linux to the address established in the LTZVisor.

The Linux version chosen was the 2015.4 Xilinx version made available by Xilinx on their Git repository. Xilinx also has available a few default configurations for their various SoCs, including for the Zynq-7000-based boards like the ZYBO. So, the first step was to apply the Zynq configurations to the Linux Kernel code. However, as aforementioned, some modification are required, since Linux will run atop the LTZVisor and as the non-secure guest. This entails modifications mainly to the Linux kernel image.

So, the first modifications executed were to the Linux device tree, or the ".dts" file. Again, following a minimal approach, only the absolutely necessary devices were maintained to be mapped in the Linux device tree, such as the global timer, interrupt controller and UART peripherals, along with the more obvious CPU, memory and SLCR mappings. Furthermore, because the Linux OS cannot use the whole of the DRAM, its start address had to be modified.

Still regarding the alterations to the Linux kernel and because of the LTZVisor's interrupt model of routing FIQs to the secure world and IRQs to the non-secure world, the FIQ stack initialization has to be removed. This will not be a problem for the Linux normal execution, since no FIQ will even reach the non-secure world.

Moreover, as Linux may need to access some secure components, even if mediated by the hypervisor layer, the ability to access those components, as well as some secure CP15 registers and some SLCR, should be given. To achieve this, three extra SMCs were added: *secure_read*, *secure_write* and *secure_cp15_write*. These allow for the non-secure guest to access some secure resources, including the CP15 and SLCR registers, under supervision of the LTZVisor. Listing 4.7 shows the added system calls.

```
1 #define __ARM_NR_BASE          (__NR_SYSCALL_BASE + 0x0f0000)
2
3 #define __ARM_NR_secure_read              (__ARM_NR_BASE + 6)
4 #define __ARM_NR_secure_write             (__ARM_NR_BASE + 7)
5 #define __ARM_NR_secure_cp15_write        (__ARM_NR_BASE + 8)
```

Listing 4.7: Added SMCs

After executing all the necessary modifications to the Linux kernel, the next step should be to compile it. To do this, the chosen toolchain was the GCC Linaro cross-compiling toolchain for the 32-bit ARMv7 architecture.

To complete the whole Linux image, we then include the rest of the components to a single image of the LTZVisor. To achieve this we used the *zcomposite* Linux bootloader, a minimal Linux bootloader that comprises all of the absolutely necessary components, the previously compiled Linux kernel, the file system and the device tree blob (generated according to the Device Tree file), into a single ".elf" binary file. This is described in the *zynq_linux_boot.lds* file, a script file containing the general offset of the whole Linux image, and the partial offsets for each individual sections. Listing 4.8 contains the *zynq_linux_boot.lds* file with each section offset. The first represented section is the previously compiled *clearreg* file, which will clear the "Filtering_Start_Address_Register" to start the address filtering at the correct memory address. Then, with the appropriate offsets, the next sections will be the device tree blob, followed by the Linux File system (ramdisk image) and the Linux Kernel.

```
1 PHY_OFFSET = 0x2000000;
2 SECTIONS {
3         . = PHY_OFFSET;
4         .texts : {
5                 _start:
6                 clearreg.o(.text);
7                 clearreg.o(.rodata);
8         }
9         . = PHY_OFFSET + 0x4000;
10        .textd : {
11                d.tmp(.data);
12                . = . + 0x1000;
13        }
14        . = PHY_OFFSET + 0x800000;
15                .textr : {
16                r.tmp(.data);
17        }
18        . = PHY_OFFSET + 0x8000;
19                .textz : {
20                _start_linux:
21                z.tmp(.data);
22        }
23 }
```

Listing 4.8: ZComposite Linux Script

Finally, this script's memory offset needs to match with the starting address of the Linux image and the start of non-secure memory portion.

4.4.2 TZ-VirtIO Integration

After having the Linux image compiled and included in the LTZVisor, the next step would be to integrate the already implemented TZ-VirtIO code [OMC+18] with this LTZVisor version, so we can have communication between the two OSes. To achieve this, the VirtIO drivers must be included both in the secure world side, and the non-secure side.

To use the VirtIO layer in the secure world, TZ-VirtIO follows the implementation of the OpenAMP project to use the VirtIO drivers with FreeRTOS. So, the aptly named "comm_lib" source folder is added to the LTZVisor directory and its path added to the Makefile for compilation alongside the LTZVisor. Figure 4.11 presents the communication library introduced in the LTZVisor source code file tree.

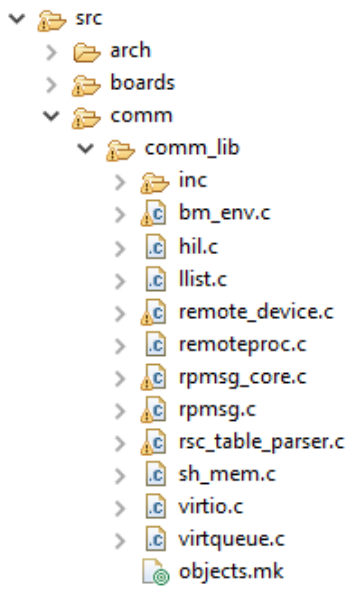


Figure 4.11: VirtIO Communication Library Folder in the LTZVisor's File Tree

As aforementioned, because the secure guest OS is compiled as part of the hypervisor layer, the FreeRTOS will be able to directly access the communication library. Furthermore, some additions to the hypervisor layer are needed. These mainly include the setup of extra interrupts for both the secure and non-secure worlds, which will be used for the notification of each side for a myriad of different circumstances.

Furthermore, some modifications were executed to the OSes source code for using the TZ-VirtIO layer for communication. Although both OSes can be the communication master in this TZ-VirtIO communication strategy, our implementation will focus on the use-case consisting of the secure world as the communication master, wanting to send information to the non-secure side, which will in turn act as the communication slave.

Regarding the secure FreeRTOS guest, since the TZ-VirtIO Communication library is already added to the LTZVisor code, the only addition needed to the FreeRTOS code will be the application that will create the master communication channels and send the message.

On the Linux side however, the VirtIO layer has to be added separately. This includes the addition of the modified RPMSG driver and the *remoteproc* files, both of which are modifications done by the TZ-VirtIO implementation to the Linux original source files. Furthermore, and contrary to the FreeRTOS implementation, the Linux OS features the slave application, that opposites the FreeRTOS master.

As similarly done with FreeRTOS, interruptions relating to the VirtIO communication layer have to be added, including an extra SMC instruction for requesting a secure world interrupt.

4.4.3 Hardware Modules Integration

The last step for attain the complete system should be the integration of the implemented hardware modules with the LTZVisor and TZ-VirtIO layers. After both guests are included, working and communicating via the TZ-VirtIO communication mechanism, the hardware mechanisms should be integrated with the system, replacing the software copy mechanism. Again, because the use-case consists on having the secure world as the communication master, the needed APIs were included in the secure world side. However, because we want the ability of interchanging between the various mechanisms, for now, the introduced code for the hardware mechanisms' APIs will be included for compilation one at a time, being possible to alternate between them through three makefile defined macros. This facilitates the switch between mechanisms while reducing the number of lines of code.

4.4.3.1 DMA IPC

To integrate the DMA module with the LTZVisor code, the "DMA.c" file, containing the DMA module API functions aforementioned in the DMA IPC module implementation section, was included in the "comm_lib" directory.

During the LTZVisor hardware boot, the DMAC is configured using the DMA initialization function, thus enabling its further use for the creation of DMA channels and transfers. The next step was to replace the "env_memcpy" function used in the *rpmsg* TZ-VirtIO drivers that executed the data transfer between source and destination addresses, with the "DMA_Transfer" function. Because the DMAC was previously configured by default, this function will only setup the DMA channel used for the transfer, receiving as parameters the same variables that the memory copy mechanism needs: source address, destination address and message length. Listing 4.9 shows the addition of the DMA Transfer method.

```
1 void env_memcpy(void *dst, void const *src, unsigned long len){
2 #ifdef DMA_IPC
3         DMA_Transfer(DmaInst, &DmaInst->Config, src, dst, len);
4 #endif
5 }
```

Listing 4.9: DMA Transfer Function Addition

Although the DMA transfer is executed by these two functions, some extra modifications need to be carried out. Unlike the software memory copy, when using an hardware-based memory copy mechanism, the CPU is freed while the message is being transferred, and the LTZVisor will follow its execution flow. However, some considerations have to be taken into account:

- The communication mechanism task must come to an halt while the message transfer is being executed, as it assumes that the transfer is complete;
- The TZ-VirtIO mechanism must notify the non-secure guests when the message is delivered;
- Synchronism must be achieved while using the hardware resources.

Taking into account these new concerns, that emerged from the replacing of the transferring mechanism, some additions committed to the *rpmsg* driver, consisting on:

- New mutex added, controlling the access to the hardware transferring mechanism, locking/unlocking whilst being used;
- Addition of a new interrupt, triggered by the hardware module.

So, there was the necessity of configuring the new hardware interrupt for the DMA transfer finish event. Listing 4.10 represents the setup of the first DMA channel interrupt, consisting on the security setting of the interrupt (set as secure to be handled by the communication master, secure world), enabling and linking the according interrupt handler.

```
1 #define DMA_Channel_1 46
2
3 uint32_t ltzvisor_hw_init(){
4         ...
5         interrupt_security_config(DMA_Channel_1, Int_S);
6         ...
7 }
8
9 void SetupDMAInterrupt(){
```

```
10          interrupt_enable(DMA_Channel_1, TRUE);
11          vFreeRTOS_handler_set(DMA_Channel_1, handler_DMA_IPC);
12 }
```

Listing 4.10: DMA Interrupt Setup

Finally, during the LTZVisor execution, and when the DMA interrupt was triggered, the "handler_DMA_IPC" function, represented in Listing 4.11, will clear the interrupt (both the DMA and GIC flags), notify the non-secure Linux guest, and give the previously taken semaphore back to the system, thus allowing for other tasks to access the DMAC and for the communication task to do its final clean-up.

```
1 void handler_DMA_IPC(uint32_t int_id){
2          // Clear the GIC Interrupot Flag
3          interrupt_clear(int_id, get_cpu_id());
4          // Clear the DMA Interrupt Flag
5          *DMA_Clean_ptr = 0xFF;
6          if(ipc_semaphore != NULL){
7                  // Notify the Linux Guest OS
8                  virtqueue_kick(rdev_1->tvq);
9          }
10         xSemaphoreGive(ipc_semaphore);
11 }
```

Listing 4.11: DMA Interrupt Handler

4.4.3.2 AXI IPC

As for the AXI IPC hardware module, its software integration should be much simpler than that of the DMA API, since no previous setup is needed because all of it is done through the hardware logic. So, like the DMA implementation, the software memory copy mechanism was replaced with the configuration of the AXI hardware module. This is done by writing to the AXI Lite port configuration registers and finishing by triggering the start of the transfer in the same fashion (Listing 4.12).

```
1 void env_memcpy(void *dst, void const *src, unsigned long len){
2 #ifdef AXI_IPC
3          *(IPC_address + 0) = len;
```

```
4         *(IPC_address + 1) = src;
5         *(IPC_address + 2) = dst;
6         // STart the transfer
7         *(IPC_address + 4) = 0x1;
8 #endif
9 }
```

Listing 4.12: AXI Transfer Function Addition

This way, the hardware transfer is completed but, like the DMA module, the same synchronism and interruptions additions must be featured. So, Listing 4.13 shows the configuration of the AXI IPC module interrupt, similar to the aforementioned DMA.

```
1 #define AXI_int 61
2
3 uint32_t ltzvisor_hw_init(){
4         ...
5         interrupt_security_config(AXI_int, Int_S);
6         ...
7 }
8
9 void SetupAXIInterrupt(){
10        interrupt_enable(AXI_int, TRUE);
11        vFreeRTOS_handler_set(AXI_int, handler_AXI_IPC);
12 }
```

Listing 4.13: AXI Interrupt Setup

Finally the AXI IPC interrupt handler will clear both GIC's interrupt flag and the hardware module's trigger register, thus resetting the module. The Linux side is also notified and the IPC mutex is unlocked for further use (Listing 4.14).

```
1 void handler_AXI_IPC(uint32_t int_id){
2         // Clear the GIC Interrupot Flag
3         interrupt_clear(int_id, get_cpu_id());
4         // Clear the DMA Interrupt Flag
5         *(IPC_Clean_ptr + 4) = 0x0;
6         if(ipc_semaphore != NULL){
7                 // Notify the Linux Guest OS
8                 virtqueue_kick(rdev_1->tvq);
9         }
```

```
10          xSemaphoreGive(ipc_semaphore);
11 }
```

Listing 4.14: AXI Interrupt Handler

5. Evaluation

In this section, the test and evaluation of the hTZ-VirtIO implementation are presented. This encompasses the explanation of how and why these tests, and comparisons with the previous implemented TZ-VirtIO software mechanism, were executed, followed by the presentation of the results, divided into each section: engineering effort, memory footprint, hardware costs and performance. Also, each section will also discuss the impact and reasons for each attained results and metrics.

To evaluate the implemented system, the hTZ-VirtIO system was deployed in the Xilinx's ZYBO board, a Zynq-7000 SoC featuring a dual ARM Cortex A9 processor, used in a single-core configuration and running at 650 MHz. Because the main goal of the project was to replace the software-based IPC mechanism with hardware modules, the executed evaluation was centred on the comparison between the various mechanisms.

5.1 Engineering Effort

The first evaluated measure was the engineering effort required for the implementation of each method. To determine this metric, the number of code lines for each strategy was used. This is a standard method of approximating the engineering effort for software, and in this case hardware, development. Although not absolute, specially when comparing between software (C and ARM Assembly language) and hardware (Verilog language) code, this estimate will sufficiently demonstrate the desired metric.

To count the Lines-of-Code (LoC) number, a static analysis tool was used: the Understand software by SciTools. By using this tool we were able to count the number of both software and hardware LoC for each of the implementations.

Figure 5.1 represents the LoC value for 4 different scenarios, none of which include the Linux OS, as it is constant and not relevant, and the number of LoC

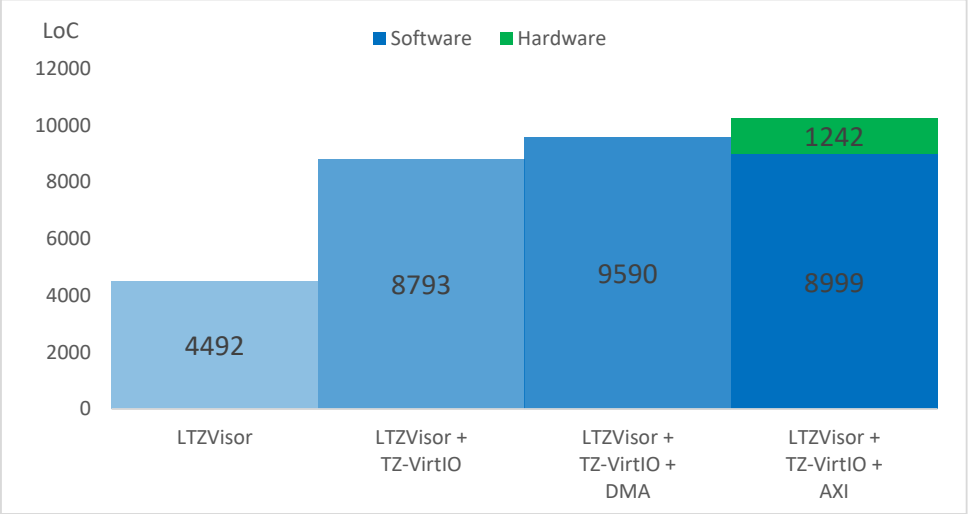


Figure 5.1: Comparison between Line-of-Code Number

would be too great, thus masking the true comparison between the mechanisms. The different scenarios are:

- LTZVisor: containing the LTZVisor and FreeRTOS code;
- LTZVisor + TZ-VirtIO: containing the same components as the LTZVisor version, but including the TZ-VirtIO communication layer;
- LTZVisor + TZ-VirtIO + DMA: containing LTZVisor, FreeRTOS, TZ-VirtIO layer and the DMA API;
- LTZVisor + TZ-VirtIO + AXI: containing the LTZVisor, FreeRTOS and the Verilog code (represented with a different colour) for the AXI IPC module.

Although not directly relevant to the context of this thesis' evaluation, the first LTZVisor evaluation (without the communication layer) was maintained as a way of demonstrating the scale of the LoC increase for each mechanism. So, as observed in the Figure 5.1 graph, the inflation of adding the communication layer is of a much bigger nature, almost 95% increase (from 4492 to 8793 LoC) than that of the addition of the hardware mechanisms.

Compared to the software implemented mechanism, the DMA IPC mechanism represented a 9% increase in LoC number. This is due to the addition of the DMA API as well as the the DMA mechanism interrupt handler. As for the AXI IPC implementation, the increase in software LoC was of only 3%, but it brought an extra 1242 Verilog LoC, thus totalling a 16% increase when compared to the default TZ-VirtIO implementation.

5.2 Memory Footprint

To obtain the memory footprint values for each of the implemented solutions, the ARM GNU toolchain's Size tool was used. This tool is able to calculate the memory overhead of an output or image file divided into three different sections:

- **.text:** containing the executable code, as well as constant variables and vector tables;
- **.data:** containing initialized system variables;
- **.bss:** containing non-initialized system variables, stack and heap dynamic variables.

Using this evaluation strategy, we were then able to compare the memory overhead of each of the implemented mechanisms. Again, for contextualization, the standalone LTZVisor layer, without the TZ-VirtIO layer was included. Unlike, the previous section, the Linux OS is included for the calculation of the ".elf" size, as for the TZ-VirtIO layer to be compiled and working, the non-secure Linux guest must be present.

	Memory Footprint in Bytes			
LTZVisor Version	.text	.data	.bss	**Total**
LTZVisor	15022382	484	460448	15483314
LTZVisor + TZ-VirtIO	15055023	1460	461136	15517619
LTZVisor + TZ-VirtIO + DMA	15065141	1476	463592	15530209
LTZVisor + TZ-VirtIO + AXI	15055583	1468	461128	15518179

Table 5.1: LTZVisor and TZ-Virtio Memory Footprint

As observed in Table 5.1, the increase brought on by the addition of the communication layer itself is of a much greater relevance than the size increase of both hardware data-transferring mechanisms. This is due to the smallness of the implemented software APIs, specially with the AXI implementation, where most of the code is Verilog code, and not part of the measured ".elf" image.

5.3 Hardware Costs

To evaluate the hardware costs of the implemented AXI IPC mechanism, we recurred to Vivado's utilization report statistics, which present the statistics for each

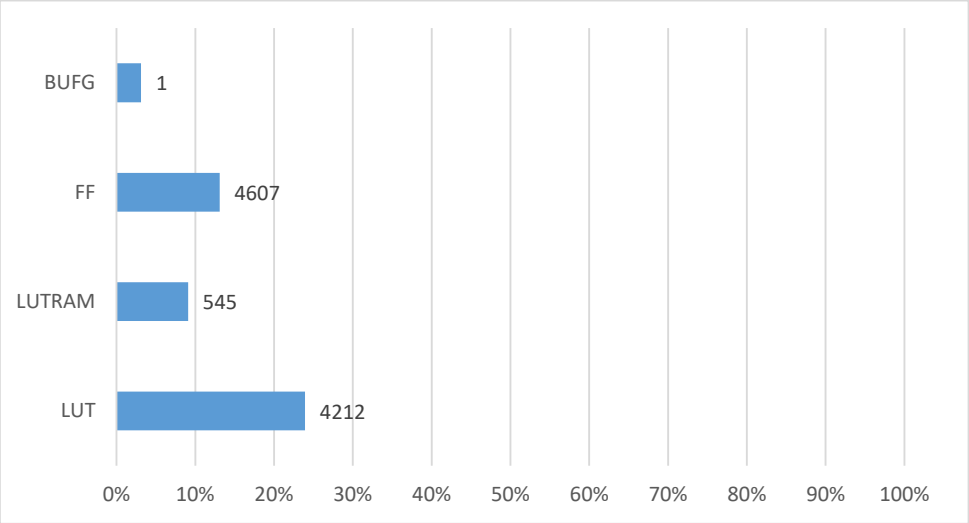


Figure 5.2: Hardware Resource Utilization Statistics

of the FPGA Configure Logic Block types. In this case, the resources' types usage consisted of Global Buffers (BUFG), Flip-Flops (FF), Look-Up Tables (LUT) and Look-Up Table RAMs (LUTRAM).

So, the post-implementation reports of the hardware resource usage for the implemented module are represented in Figure 5.2, with each block type showing its usage percentage of that specific hardware resource available on the ZYBO board, as well as its definite number value. These results are presented in further detail in Table 5.2.

Block Type	Utilization (nr)	Available (nr)	Percentage (%)
BUFG	1	32	3.13
FF	4607	35200	13.09
LUTRAM	545	6000	9.09
LUT	4212	17600	23.93

Table 5.2: Hardware Resource Utilization Statistics

Although these results cannot be directly comparable to any of the other implementations, since both do not implement FPGA logic, we can say that implementation still fits the solution, as it does not entail much hardware costs. All of the used resources were far from occupying half of the FPGA fabrics on the ZYBO board FPGA fabrics, which is the entry board for the Zynq-7000 line of SoCs.

5.4 Performance

To measure the various metrics related to the system performance, the Performance Monitoring Unit (PMU) present on the ZYBO board was used. The PMU was used in its cycle counter mode, which is capable of counting every processor clock cycle. So, the PMU is used to count how many clock cycles a certain operation, that is intended to be measured, takes. Because, as aforementioned, the ZYBO board's CPU clock is set to 650 MHz, each clock cycle will take 1,53 nanoseconds. Thus, at each 1,53 nanoseconds, the PMU will count one cycle.

5.4.1 Baremetal

Before evaluating the performance of the hardware mechanisms integrated in the TZ-VirtIO layer, we must first measure its baremetal performance as a data transfer mechanism. This will demonstrate their standalone performance and data throughput before integrated with the Hypervisor and TZ-VirtIO layer.

Figure 5.3 presents the data throughput for each of the mechanisms that are used in the TZ-VirtIO and hTZ-VirtIO systems. The data transfer length was varied from 16 to 65536 bytes, duplicating with each data step increase. This allows for each mechanism behaviour to be evaluated when the data length is being varied. Observing the resulting graph (Figure 5.3), we can verify that the software copy mechanism (using a *memcpy* function) is mostly constant and presenting a much smaller data throughput than both of the hardware transfer mechanisms, capping at 150Kbps. For the smaller data sizes however (less than 32 bytes transferred), the software mechanism presents a marginally better throughput.

As for the AXI mechanism, its data throughput increases exponentially. However, the throughput peaks at 1.38 Gbps when the transfer size is of 1024 bytes. This is due to the maximum burst transfer of the AMBA Interconnect being of 256 words of 32 bits each, which means that, when the transfer length is bigger than 1024 bytes, for each 256 extra words, the hardware transfer has to re-triggered. Because of this, the data throughput when using this mechanism drops significantly when above a 1024 bytes transfer, stabilizing at a 775 Kbps data throughput.

Finally, the DMA implementation presents a much lesser data throughput for smaller message sizes, but increases greatly after a 1024 bytes transfer size, value at which it has bigger throughput than that of the software mechanism. When compared to the AXI mechanism, the DMA will have a bigger throughput for any transfer size bigger than 8192 bytes.

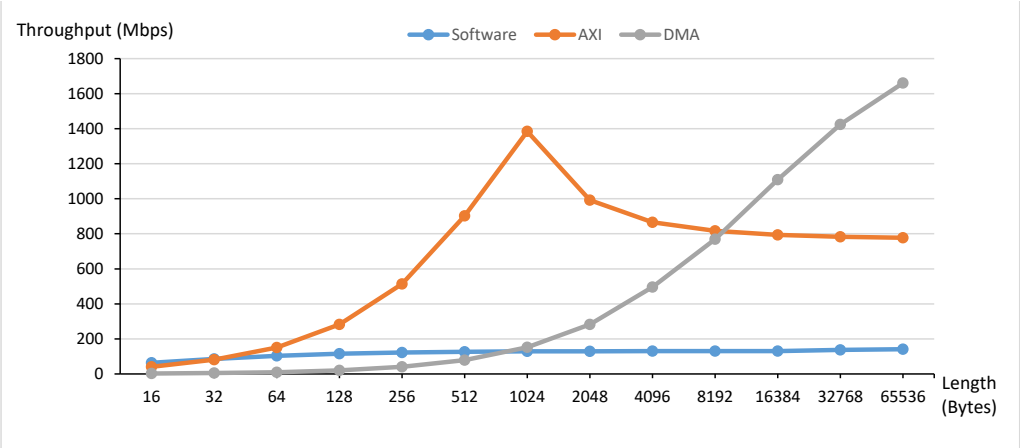


Figure 5.3: Baremetal Throughput Comparison

So, in a baremetal application, each software mechanisms will present advantage regarding transfer sizes of:

- Software mechanism: any transfer size smaller than 32 bytes;
- AXI mechanism: from 32 to 8192 bytes transfer size;
- DMA mechanism: transfer lengths bigger than 8192 bytes.

5.4.2 hTZ-VirtIO

To properly compare the performance of the various transfer mechanisms integrated in the TZ-VirtIO layer, this evaluation scenario followed the best case estimate strategy. We chose this strategy as the only of way of effectively comparing the various mechanisms, since having performance metrics measured in any real use-case scenario would be influenced by a myriad of different external factors, including:

- **Number of RTOS tasks:** any increase in the number of tasks executing in the secure FreeRTOS, regardless of its priority level, will most definitely interfere with the overall execution flow of the communication mechanism, thus leading to performance degradation. Any task with lower priority than the communication task will add latency between the sending of the message and the notification of the non-secure OS, as this notification only occurs at a world switch, which in turn only occurs in the FreeRTOS idle times. Any task with an higher interrupt level will execute before the communication task, and thus delaying the communication;

- **Number of Linux Applications:** like the FreeRTOS, the Linux communication response time will also vary depending on the number of executing applications;

- **FIQs:** if a FIQ occurs during the communication task, the communication performance will undoubtedly be affected. Likewise, the FreeRTOS tick frequency may entail communication overhead.

So, to remove any unwanted interference that would jeopardize the comparison between the mechanisms, our evaluation strategy removed any other real-time task while maintaining only the communication task. As aforementioned, the communication master is always the higher priority FreeRTOS, as it is the secure OS that schedules the non-secure Linux OS, which is notified at a context switch. Furthermore, because of the buffer size limitation of the TZ-VirtIO system, the data throughput was varied in relation to the message size, and also the number of messages sent.

Like the previously mentioned evaluation, to measure both the data throughput and message latency for the different message sizes, the PMU was used. To measure the data throughput, the PMU was ordered to start measuring just before the message transfer is started by the secure OS and stopped when the Linux OS receives the whole message. Regarding the latency, the PMU measured the time that the FreeRTOS took from the start of the communication application, to the freeing of the CPU core. This shows the time that the secure OS is busy in the communication task for each mechanisms.

5.4.2.1 Message Size Variation

First, for each mechanism, only the message size was varied, while only sending one message between each context switch. This variation was made from 16 to 1024 bytes and both the throughput and latency were measured for each case.

As a reference for comparison, the measured results regarding the software transfer mechanism are shown in Figure 5.4. Both latency and throughput increase gradually with the increase of the message size, maxing out at 133 μs and 21 Mbps, respectively, for a message size of 1024 bytes.

As for the hTZ-VirtIO transfer mechanisms, both the AXI and DMA mechanisms present a constant latency time when the message size varies, even if at much different levels, since the AXI IPC latency is constant at 22 μs, and the DMA IPC is at a much higher latency time of 1.8 ms. This is due to the data transfer itself being made through the hardware layer, and the setup time not depending

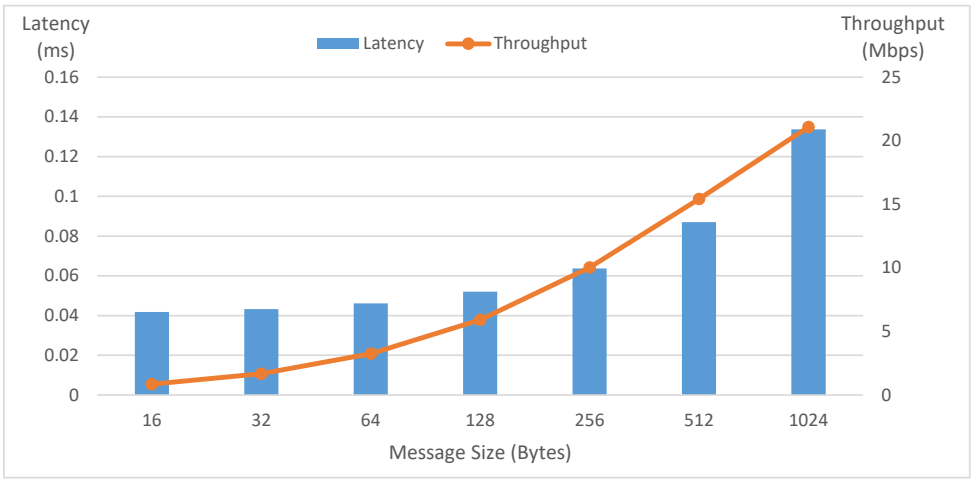
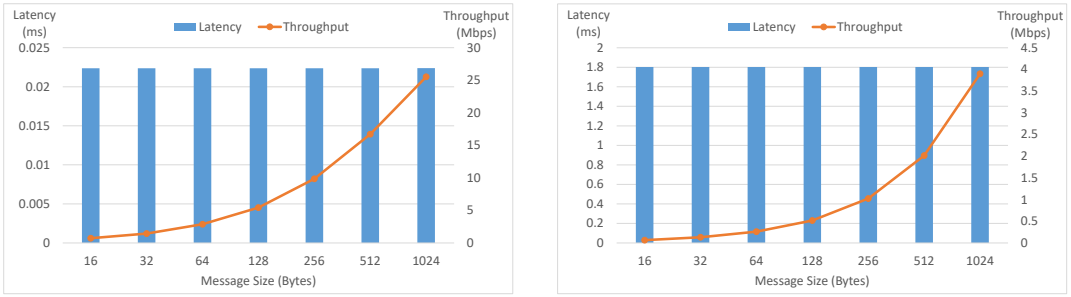


Figure 5.4: Software Message Size Variation Performance



(a) AXI Message Size Variation Performance

(b) DMA Message Size Variation Performance

Figure 5.5: hTZ-Virtio Message Size Variation Performance

on the size of the message. As for the throughput, its rise is exponential but, like the latency metric, at very different values, with the maximum throughput for both mechanisms being of 25 Mbps for the hardware implementation, and 3.9 Mbps for the DMA implementation. Both results are displayed in Figure 5.5.

Figure 5.6 presents the comparison between the software and AXI IPC transfer mechanisms for both latency and throughput measurements. The DMA IPC mechanism was removed from the comparison as its latency magnitude is too large, and throughput magnitude too small, when compared to the Software and AXI IPC mechanisms. When only the message size is varied, the data throughput of both mechanisms will be closer, with a small advantage to the AXI IPC mechanism, while the latency will be constant and much smaller in the AXI IPC method, thus giving it an advantage. This means that in this scenario, although the message won't arrive much quicker to the non-secure OS, the secure OS performance will be much greater when using the AXI IPC mechanism, as the software overhead, and thus performance overhead, will be much lesser when compared to

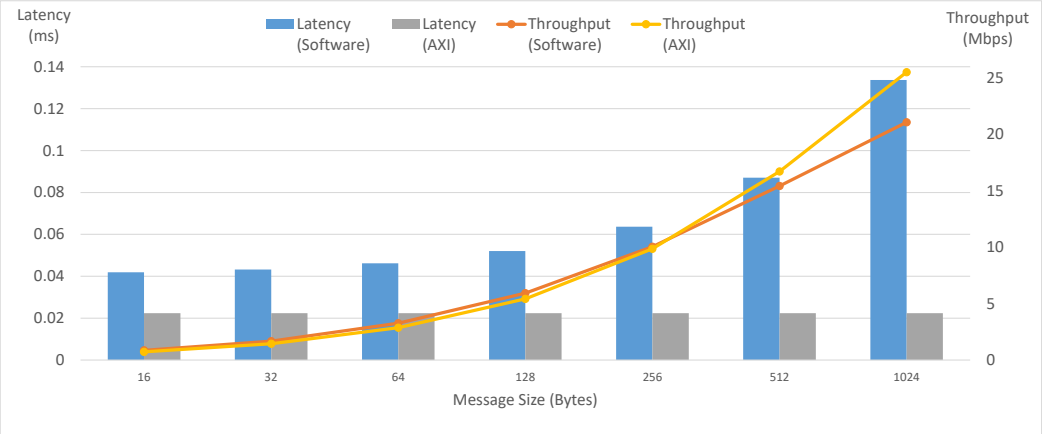


Figure 5.6: Message Size Variation Performance Comparison

the software-based implementation.

5.4.2.2 Message Number Variation

After testing and evaluating the behaviour of the various mechanisms by varying the TZ-VirtIO message size, the next step would be to evaluate the performance, latency and data throughput, when the number of messages is varied. To do this, the message size was kept at 256 words, or 1024 bytes, for bigger throughput and lower latency, while the number of messages was varied between 1 and 64 messages, effectively varying the data length from 1024 bytes to 65536 bytes. By doing this, we can effectively evaluate and compare the various mechanisms for bigger transfer sizes.

As with the previous tests, both the data throughput and latency were measured for each mechanisms and while varying the total transfer length. Because the number of messages is being increased, the notification of the non-secure side only occurs when all messages are sent, and not with each message. This means that the throughput and latency measurements will measure the total time from starting the sending of the messages, to when the Linux receives all the sent messages.

Like before, the software mechanism was measured for comparison, and its results are present in Figure 5.7. Because in this test the number of messages is being doubled for each variation, the latency of the software mechanism is also doubling, reaching a value of 8.5 ms for a transfer size of 64 KB. For the same reason, the data throughput only has a small increase, stabilizing at around 25 Mbps.

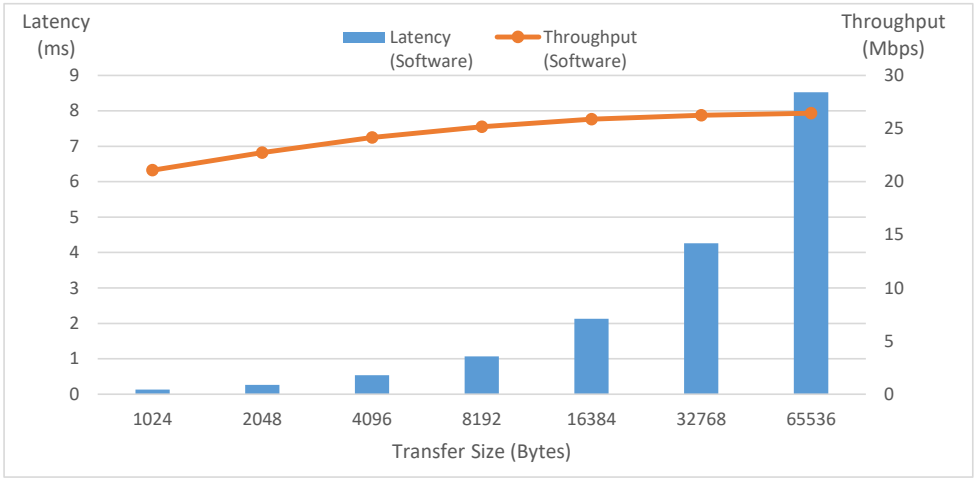
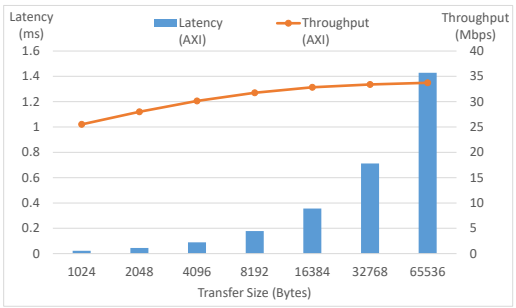
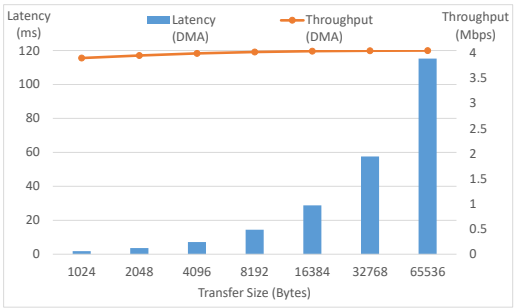


Figure 5.7: Software Message Number Variation Performance

As for the hTZ-VirtIO transfer mechanisms, its performance is presented in Figure 5.8. Regarding the AXI IPC mechanism, represented in Figure 5.8a, we can verify that the data throughput does not improve greatly with the increase of the transfer size. This is due to the messages being doubled as a way of increasing the transfer size, and not an increase in the message size. Because of the AMBA interface limitations, the transfer improvement will only be as great as the biggest standalone message of 1024 bytes. The same applies for latency, where the latency will double consistently when the number of message doubles. For a transfer size of 65536 Bytes (64 messages of 1024 Bytes), the data throughput reaches transfer speeds of 33 Mbps, while the latency is of around 1.4 ms.



(a) AXI Message Number Variation Performance



(b) DMA Message Number Variation Performance

Figure 5.8: hTZ-Virtio Message Number Variation Performance

Regarding the DMA IPC implementation, the performance variation for both latency and data throughput metrics shows a similar evolution to that of the AXI IPC evaluation, showing almost no increase in data throughput, while doubling the latency with each transfer size duplication. However, and just like the evaluation

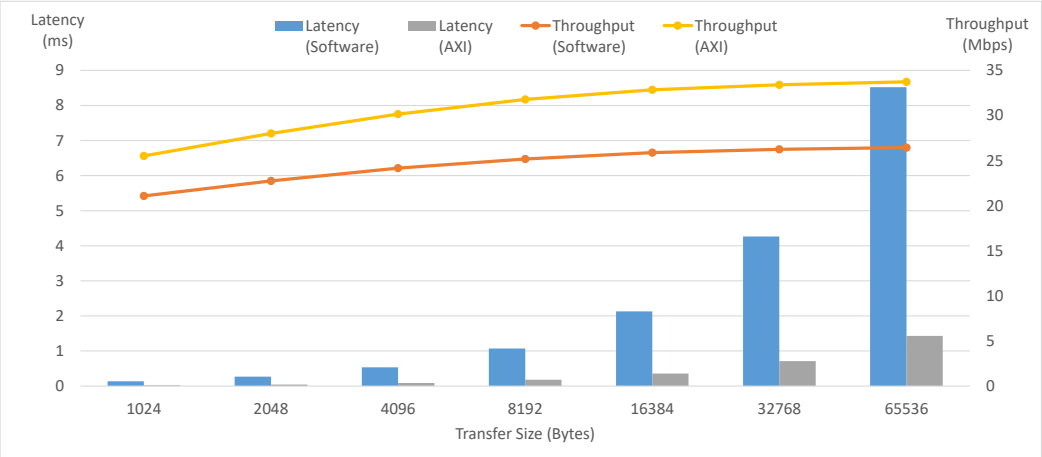


Figure 5.9: Message Number Variation Performance Comparison

executed for the variation of only the message size, the magnitude of both latency and data throughput performance is at a very different level than that of the software and AXI IPC mechanisms, with the latency reaching values of 115 ms and the throughput maxing out at only 4 Mbps.

Finally, Figure 5.9 presents the comparison between the software and AXI IPC mechanism. As with the previous evaluation, the DMA measured values were removed for being too outlandish and at a much different level than those of the other mechanisms, not allowing a good comparison between the two remaining mechanism. Regarding the comparison between the software and AXI mechanisms, the difference between the two is noticeable but not massive, with the former maxing out at 26 Mbps and the latter maxing out at 33 Mbps, representing a 26% increase in data throughput for the maximum evaluated transfer size of 65536 Bytes. Regarding the latency however, the AXI IPC presents a much better performance than the software mechanism, as the AXI latency results are consistently 6 times smaller than the software implemented mechanism. As aforementioned, this will result in much better FreeRTOS performance, as it will incur much less software overhead.

5.5 Discussion

Taking all the above results into account, we can draw some conclusions regarding the various metric comparisons between the mechanisms, even if some metrics should be taken into account with different degrees of importance. Because all metrics were based on the best case scenario for the functioning of the IPC mechanism (this only applies to some metrics), we are able to get a true

comparison between the mechanisms and thus take conclusions in regards to its practicality for this system.

First off, we can promptly verify that the DMA-based IPC mechanism implementation is not a suitable contender when integrated in the TZ-VirtIO IPC mechanism. This is undoubtedly because of TZ-VirtIO's buffer size limitations. With the first baremetal test, we can verify that for the maximum buffer size of the TZ-VirtIO system, the DMA IPC*s throughput is just as high as the software mechanism. As such, when added to the TZ-VirtIO layer, the added latency proves to be decisive for the DMA IPC mechanism, and its performance degrades significantly. The latency increases aggressively and, consequently, the data throughput decreases in the same order, even going as far as degrading the performance of the TZ-VirtIO system.

However, although not suitable for the TZ-VirtIO IPC implementation, this does not invalidate the use of DMA for any other IPC mechanism. In fact, the use of DMA-capable mechanisms for IPC has been tried and trusted for a myriad of systems, and the performance results of the baremetal evaluation, as well as the small engineering effort and memory footprint added, can prove precisely that fact. For bigger message sizes, the DMA mechanism throughput is much greater than that of the other mechanisms, and because of its implementation, the throughput will reach its maximum capacity and much higher values. This means that the throughput will most certainly dwarf the added latency, and make such implementation viable.

Regarding the AXI IPC mechanism, when evaluated in a baremetal data transferring approach, we can verify that its maximum throughput peak occurs at the maximum buffer size of the TZ-VirtIO system. This bodes well for the implementation mechanism, and it is in fact verified by the hTZ-VirtIO measurements. Because its setup latency is so low, and its data throughput so high, when compared to the software mechanism, especially when using the maximum message size, the AXI IPC implementation suits the TZ-VirtIO mechanism perfectly. The integration of this mechanism with the TZ-VirtIO system will not only increase the data throughput, but also, and perhaps more importantly, reduce the latency and software overhead. This is crucial in any critical system, as the secure, critical OS will spend much less time in the communication tasks, thus maintaining its real-time capabilities. This translates into an overall better mechanism and the best implementation for the LTZVisor's IPC mechanism.

There are some drawbacks nonetheless, mainly the added hardware costs. Although the added engineering efforts are minimal, and the memory footprint completely negligible, implementing an FPGA hardware module entails hardware costs which must be taken into account. However, the hardware costs are still minimal, only occupying a maximum of 25% on some FPGA fabrics on the ZYBO board, which means that this solution is feasible on most low-end devices, and well worth the trade-off between performance and hardware-costs.

6. Conclusion

The use of virtualization technology and hardware-offloading strategies on embedded systems is definitely not a new concept. In fact, both have been raved about consensually in the scientific community for their many benefits on solving the problems that emerge when trying to keep up with the fast-growing needs of the market and its consumers. The rise in complexity and dependability on safety and security critical embedded systems is aided by the concept of IoT, where many of these systems need to connect to some kind of network, thus making them more prone to external attacks and of even greater complexity.

The use of any of these strategies as a standalone solution for the aforementioned problems became unfeasible for many situations. To keep up with demand, many embedded solutions started combining more than one of these strategies. SoC manufacturers developed hardware-based virtualization extensions and FPGA components, which led system designers to exploit the hardware's maximum potential, incorporating hardware-assisted virtualization and reconfigurable hardware modules. This allowed the consolidation of multiple, very complex virtual machines into the same hardware platform without any kind of performance degradation, by the means of an underlying monitoring layer, the hypervisor.

Because of the desire to completely encapsulate these systems as a way of providing isolation and fault containment, these virtualized system, although sharing the same hardware resources, are not able to communicate in any other way other than with some *Over-the-Air*, for example, TCP/IP communication protocol. Undoubtedly, the use of TCP/IP communication in any critical, real-time system is completely ludicrous, especially due to the sheer size of the network stack of this protocol, meaning a great deal of software overhead and latency, which would most assuredly lead to the lost of the system's real-time capabilities. Instead, most hypervisors implement an IPC mechanism, thus enabling communication via the hypervisor layer, which greatly reduces the software overhead (when compared to the TCP/IP network stack). However, in some cases the small software overhead introduced by the addition of the IPC mechanism might be sufficient to affect the

best functioning of the system.

So, this thesis proposes the hTZ-VirtIO IPC mechanism, an hardware-based communication mechanism that consists on the offloading of the software-based IPC mechanism for the LTZVisor, an in-house, open-source, TrustZone-assisted hypervisor. This project focuses on the implementation of two different hardware alternatives, as well as their integration in TZ-VirtIO, the VirtIO-based, IPC mechanism of the LTZVisor. hTZ-VirtIO consists on the DMA IPC and AXI IPC mechanisms. The first uses the on-board DMAC controller to create DMA channels that replace the software data transfer of the messaging mechanism. The latter, AXI IPC mechanism, consists on a software-programmable reconfigurable hardware module that uses the AMBA interfaces and the AXI protocol for also replacing the software-based communication mechanism of the TZ-VirtIO. This hardware-offloading strategy provides reduction of latency and software overhead whilst increasing data throughput, thus improving the system's real-time capabilities with the small added cost of the hardware resources. Furthermore, because TrustZone's security oriented hardware extensions extend the physical world separation to the peripherals, FPGA and memory, the isolation and fault containment of the various machines is maintained, as well as preserving safety and security capabilities.

Although the expected benefits of hardware-offloading were not attained with the DMA IPC mechanism, the hTZ-VirtIO AXI IPC mechanism performs as expected, greatly reducing software latency, increasing the mechanisms data throughput, improving the real-time capabilities of the system whilst maintaining the standards of the VirtIO system and the security of the TrustZone architecture.

6.1 Future Work

This thesis effectively implements an hardware-based IPC mechanism for the LTZVisor as initially proposed. However, only one of the proposed mechanisms adequately achieves the desired goals, as the DMA-based implementation did not fulfil all of the established requirements. So, as a first step, further research could focus on executing modifications to the DMA IPC mechanism and the TZ-VirtIO layer to efficiently integrate the DMA IPC mechanism in this implementation. For instance, the setup time of the DMA channel, which is the main reason for the added latency, can be significantly diminished by fixating the core behaviour of the DMA program. Removing some possible configurations of the DMA transfer mechanism, like the transfer size, burst size and length, and setting them as default

parameters, the DMA channel setup time could be greatly reduced, thus improving in both latency and data throughput metrics.

Furthermore, future research could focus on the porting of these communication systems to other platforms, meaning both a switch of hardware or software platform. For example, the hardware AXI and DMA mechanisms could be ported to the μRTZVisor, another in-house TrustZone-assisted hypervisor, with a microkernel-like implementation and multi guest OS support. Because this hypervisor was also implemented in a Zynq-7000 SoC, the porting of the hardware mechanisms would only consist on the integration with the hypervisor itself. This could bring great advantages to the hypervisor as, being a microkernel-like implementation, the IPC mechanism is of an even greater importance, and reducing its latency and enhancing its throughput would surely boost the performance of the hypervisor [RSTP18].

On the other way around, the implemented mechanisms could also be transferred to other systems, even systems that do not implement virtualization. Any FPGA capable board would be able to hold the AXI IPC module, and this implementation could be used for any type of system that relies on fast, reliable and big data transfers.

References

[ABYY10] N. Amit, M. Ben-Yehuda, and B. A. Yassour. IOMMU: Strategies for Mitigating the IOTLB Bottleneck. In *International Symposium on Computer Architecture*, pages 256–274, June 2010.

[ARM09] ARM. ARM Security Technology: Building a Secure System using TrustZone Technology ARM. In *ARM White Paper*, 2009.

[ARM11] ARM. AXI and ACE Protocol Specification. Technical report, 2011.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SIGOPS Oper. Syst. Rev.*, volume 37, pages 164–177, December 2003.

[BR16] F. Baum and A. Raghuraman. Making Full Use of Emerging ARM-based Heterogeneous Multicore SoCs. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, January 2016.

[CH02] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. In *ACM Computing Surveys*, volume 34, pages 171–210, June 2002.

[CI06] Airlines Electronic Engineering Committee and Aeronautical Radio Inc. *Avionics Application Software Standard Interface: ARINC Specification 653P1-2.* Aeronautical Radio, 2006.

[DFK+07] N. Dave, K. Fleming, M. King, M. Pellauer, and M. Vijayaraghavan. Hardware Acceleration of Matrix Multiplication on a Xilinx FPGA. In *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*, pages 97–100, May 2007.

[DN14] C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 333–348, March 2014.

[DPNJ08] F. Diakhaté, M. Perache, R. Namyst, and H. Jourdren. Efficient Shared Memory Message Passing for Inter-VM Communications. In *European Conference on Parallel Processing*, pages 53–62, August 2008.

[EBTB63] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel Processing in a Restructurable Computer System. In *IEEE Transactions on Electronic Computers*, volume EC-12, pages 747–755, December 1963.

[EH13] K. Elphinstone and G. Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150, November 2013.

[GGM+16] P. Garcia, T. Gomes, J. Monteiro, A. Tavares, and M. Ekpanyapong. On-Chip Message Passing Sub-System for Embedded Inter-Domain Communication. In *IEEE Computer Architecture Letters*, volume 15, pages 33–36, January 2016.

[GGS+14] P. Garcia, T. Gomes, F. Salgado, J. Monteiro, and A. Tavares. Towards Hardware Embedded Virtualization Technology: Architectural Enhancements to an ARM SoC. In *SIGBED Review*, volume 11, pages 45–47, September 2014.

[GKS94] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC Performance for Shared-Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume 1, December 1994.

[GPG+15] T. Gomes, S. Pinto, T. Gomes, A. Tavares, and J. Cabral. Towards an FPGA-based Edge Device for the Internet of Things. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–4, September 2015.

[GSP+16] T. Gomes, F. Salgado, S. Pinto, J. Cabral, and A. Tavares. Towards an FPGA-based Network Layer Filter for the Internet of Things Edge Devices. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4, September 2016.

[HE16] G. Heiser and K. Elphinstone. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. In *ACM Trans. Comput. Syst.*,

volume 34, pages 1:1–1:29, April 2016.

[Hei08] G. Heiser. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, pages 11–16, April 2008.

[Hei09] G. Heiser. Hypervisors for Consumer Electronics. In *2009 6th IEEE Consumer Communications and Networking Conference*, pages 1–5, January 2009.

[Hei11] G. Heiser. Virtualizing Embedded Systems: Why Bother? In *Proceedings of the 48th Design Automation Conference*, pages 901–905, June 2011.

[HEK+07] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level. In *SIGOPS Operating Systems Review*, volume 41, pages 3–11, July 2007.

[HL10] G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, pages 19–24, August 2010.

[ISM09] A. Iqbal, N. Sadeque, and R. I. Mutia. An Overview of Microkernel, Hypervisor and Microvisor Virtualization Approaches for Embedded Systems. In *Report, Department of Electrical and Information Technology, Lund University, Sweden*, volume 2110, page 15, 2009.

[JSCP05] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 184–191, June 2005.

[Kai08] R. Kaiser. Alternatives for Scheduling Virtual Machines in Real-Time Embedded Systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems - IIES '08*, pages 5–10, April 2008.

[Kai09] R. Kaiser. Complex Embedded Systems - A Case for Virtualization. In *2009 Seventh Workshop on Intelligent solutions in Embedded Systems*, pages 135–140, June 2009.

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell,

H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, pages 207–220, October 2009.

[KGG06] Y. E. Kurdi, W. J. Gross, and D. Giannacopoulos. Hardware Acceleration for Finite Element Electromagnetics: Efficient Sparse Matrix Floating-Point Computations with Field Programmable Gate Arrays. In *2006 12th Biennial IEEE Conference on Electromagnetic Field Computation*, pages 397–397, April 2006.

[KLJ+13] S. W. Kim, C. Lee, M. Jeon, H. Y. Kwon, H. W. Lee, and C. Yoo. Secure Device Access for Automotive Software. In *2013 International Conference on Connected Vehicles and Expo (ICCVE)*, pages 177–181, December 2013.

[LCP+17] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, and D. Raho. VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76, pages 6:1–6:18, June 2017.

[Lie93a] J. Liedtke. A Persistent System in Real Use-Experiences of the First 13 Years. In *Proceedings Third International Workshop on Object Orientation in Operating Systems*, pages 2–11, December 1993.

[Lie93b] J. Liedtke. Improving IPC by Kernel Design. In *SIGOPS Operating Systems Review*, volume 27, pages 175–188, December 1993.

[LIJ97] J. Liedtke, N. Islam, and T. Jaeger. Preventing Denial-of-Service Attacks on a ţ-Kernel for WebOSes. In *The Sixth Workshop on Hot Topics in Operating Systems*, pages 73–79, May 1997.

[LLK08] J. Littlefield-Lawwill and L. Kinnan. System Considerations for Robust Time and Space Partitioning in Integrated Modular Avionics. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, October 2008.

[MAC+17] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto. μRTZVisor: A Secure and Safe Real-Time Hypervisor. In *Electronics*, volume 6, October 2017.

[MN11] R. Mijat and A. Nightingale. Virtualization is Coming to a Platform Near You. In *ARM White Paper*, volume 20, 2011.

[NSL+06] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor

Virtualization. In *Intel Technology Journal*, volume 10, August 2006.

[OMC+18] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto. TZ-VirtIO: Enabling Standardized Inter-Partition Communication in a Trustzone-Assisted Hypervisor. In *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, pages 708–713, June 2018.

[PGLA15] S. Patni, J. George, P. Lahoti, and J. Abraham. A Zero-Copy Fast Channel for Inter-Guest and Guest-Host Communication Using VirtIO-Serial. In *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, pages 6–9, September 2015.

[PGP+17] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares. IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices. In *IEEE Internet Computing*, volume 21, pages 40–47, January 2017.

[Pin18] S Pinto. LTZVisor: a Lightweight TrustZone-assisted Hypervisor, GitHub. [Online]. Available: `https://github.com/tzvisor/ltzvisor`, Accessed on: Sep. 9, 2018.

[POP+17] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares. Lightweight Multicore Virtualization Architecture Exploiting ARM TrustZone. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 3562–3567, October 2017.

[PPG+17] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. LTZVisor: TrustZone is the Key. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76, pages 4:1–4:22, June 2017.

[PS] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. In *ACM Computing Surveys*, volume preprint.

[PTM16] S Pinto, A Tavares, and S Montenegro. Hypervisor for Real Time Space Applications. In *The 4S Symposium*, June 2016.

[RLZ+16] Y. Ren, L. Liu, Q. Zhang, Q. Wu, J. Guan, J. Kong, H. Dai, and L. Shao. Shared-Memory Optimizations for Inter-Virtual-Machine Communication. In *ACM Computing Surveys*, volume 48, pages 49:1–49:42, February 2016.

[RM14] D. Reinhardt and G. Morgan. An Embedded Hypervisor for Safety-Relevant Automotive E/E-Systems. In *9th IEEE International Symposium on Industrial Embedded Systems*, pages 189–198, June 2014.

[RN93] D. S. Ritchie and G. W. Neufeld. User Level IPC and Device Management in the Raven Kernel. In *USENIX Microkernels and Other Kernel Architectures Symposium*, pages 111–126, September 1993.

[RSTP18] J. Ribeiro, N. Silva, A. Tavares, and S. Pinto. A TrustZone-assisted Hypervisor Supporting Dynamic Partial Reconfiguration. In *XIV Jornadas sobre Sistemas Reconfiguráveis*, pages 8 – 11, February 2018.

[Rus08] R. Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 95–103, July 2008.

[SFS96] J. S. Shapiro, D. J. Farber, and J. M. Smith. The Measured Performance of a Fast Local IPC. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operation Systems*, pages 89–94, October 1996.

[SG12] O. Schwarz and C. Gehrmann. Securing DMA Through Virtualization. In *COMPENG 2012*, 2012.

[Sha03] J. S. Shapiro. Vulnerabilities in Synchronous IPC Designs. In *2003 Symposium on Security and Privacy*, pages 251–262, May 2003.

[SHT10] D. Sangorrín, S. Honda, and H. Takada. Dual Operating System Architecture for Real-Time Embedded Systems. In *Journal of Chemical Information and Modeling*, volume 53, pages 6–15, July 2010.

[SHT12] D. Sangorrín, S. Honda, and H. Takada. Reliable Device Sharing Mechanisms for Dual-OS Embedded Trusted Computing. In *International Conference on Trust and Trustworthy Computing*, pages 74–91, June 2012.

[SHT13] D. Sangorrín, S. Honda, and H. Takada. Reliable and Efficient Dual-OS Communications for Real-Time Embedded Virtualization. In *Information and Media Technologies*, volume 8, pages 1–17, August 2013.

[TDL+12] A. Tavares, A. Dídimo, T. Lobo, P. Cardoso, J. Cabral, and S. Montenegro. Rodosvisor An ARINC 653 Quasi-Compliant Hypervisor: CPU, Memory and I/O Virtualization. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, pages 1–10, September 2012.

[TGB+08] A. S. Tanenbaum, B. Gras, H. Bos, P. Homburg, and J. N. Herder.

Countering IPC Threats in Multiserver Operating Systems (A Fundamental Requirement for Dependability). In *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing(PRDC)*, volume 00, pages 112–121, December 2008.

[UNR+05] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. In *Computer*, volume 38, pages 48–56, May 2005.

[Van10] S. H. VanderLeest. ARINC 653 Hypervisor. In *29th Digital Avionics Systems Conference*, pages 5.E.2–1–5.E.2–20, October 2010.

[Xil16] Xilinx. Zynq-7000 SoC Technical Reference Manual. Technical report, 2016.

[XPN15a] T. Xia, J. Prevotet, and F. Nouvel. Mini-NOVA: A Lightweight ARM-based Virtualization Microkernel Supporting Dynamic Partial Reconfiguration. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 71–80, May 2015.

[XPN15b] T. Xia, J. C. Prévotet, and F. Nouvel. An ARM-based Microkernel on Reconfigurable Zynq-7000 Platform. In *Mediterranean Telecommunication Journal*, volume 5, pages 109–115, April 2015.

[YBYW10] B. A. Yassour, M. Ben-Yehuda, and O. Wasserman. On the DMA Mapping Problem in Direct Device Assignment. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, pages 18:1–18:12, May 2010.

[YSA+06] D. Yaozu, L. Shaofan, M. Asit, N. Jun, T. Kun, X. Xuefei, Y. Fred, and Y. Wilfred. Extending Xen with Intel Virtualization Technology. In *Intel Technology Journal*, volume 10, pages 193 – 203, August 2006.

[Zem02] P. Zemcik. Hardware Acceleration of Graphics and Imaging Algorithms Using FPGAs. In *Proceedings of the 18th Spring Conference on Computer Graphics*, pages 25–32, April 2002.